1251941

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

November 22, 2004

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE.

APPLICATION NUMBER: *60/509,581*
FILING DATE: *October 08, 2003*
RELATED PCT APPLICATION NUMBER: *PCT/US04/33253*

BEST AVAILABLE COPY

Certified by

UNITED STATES PATENT AND TRADEMARK OFFICE
DEPARTMENT OF COMMERCE

Jon W Dudas

Acting Under Secretary of Commerce
for Intellectual Property
and Acting Director of the U.S.
Patent and Trademark Office

PTO/SB/16 (2-98) [Unisys Corp.]
Approved for Use through 1/31/2001. OMB 0651-0037
Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c).

INVENTOR(S)

Given Name (first and middle [if any])	Family Name or Surname	Residence (City and either State or Foreign Country)
John	Landis	Malvern, Pennsylvania
Terrence	Powderly	Malvern, Pennsylvania

☒ *Additional inventors are being named on the 1 separately numbered sheets attached hereto*

TITLE OF THE INVENTION (280 characters max)

Computer System Virtualization

CORRESPONDENCE ADDRESS

Direct all correspondence to:

□ Customer Number 27276

Type Customer Number here

OR

□ Firm *or* Individual Name	Unisys Corporation Michael B. Atlass				
Address	M.S. E8-114				
Address	Unisys Way				
City	Blue Bell	State	PA	ZIP	19424
Country	USA	Telephone	(215) 986-4111	Fax	(215) 986-3090

ENCLOSED APPLICATION PARTS *(check all that apply)*

☒ Specification *Number of Pages* 220

□ Small Entity Statement

□ Drawing(s) *Number of Sheets* ____

□ Other (specify) ____

METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT (check one)

□ A check or money order is enclosed to cover the filing fees

☒ The Commissioner is hereby authorized to charge filing fees or credit any overpayment to Deposit Account Number: 19-3790

FILING FEE AMOUNT ($)
$160.00

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

___ No.

___ Yes, the name of the U.S. Government agency and the Government contract number are: ____________

Respectfully submitted,

SIGNATURE ____________

TYPED or PRINTED NAME Michael B. Atlass

TELEPHONE (215) 986-4111

Dated: October 8, 2003

Registration No. 30,606

Docket Number: TN333

PROVISIONAL APPLICATION FILING ONLY

EXPRESS MAIL Mailing Label Number: EU647807493US
Date of Deposit: 10/8/03

I Hereby certify that this paper and fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" Mail Stop Provisional Application, P. O. Box 1450, Alexandria, VA 22313-1450

Michele G. McDevitt

Michele G. McDevitt

ADDITIONAL INVENTORS FOR TN333

Given Name (first and middle [if any])	Family Name or Surname	Residence (City and either State or Foreign Country)
Raj	Subrahmanian	Malvern, Pennsylvania
Aravindh	Puthiyaparambil	Malvern, Pennsylvania

CERTIFICATE OF MAILING BY "EXPRESS MAIL" (37 CFR 1.10)

Applicant(s):

DOCKET NO.
TN333

Serial No. **To Be Assigned**	Filing Date **Herewith**	Examiner **To Be Assigned**	Group Art Unit **T Be Assigned**

Invention: COMPUTER SYSTEM VIRTUALIZATION

I hereby certify that following:

Certificate of Mailing by Express Mail (1 page)

Provisional Patent Application Transmittal (3 pages)

Provisional Patent Application/Specification (220 pages)

Self-Addressed Return Post Card for Receipt (1 card)

is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 in an envelope addressed to: Director for Patents, P.O. Box 1450, Alexandria, VA 22313-14150 on October 8, 2003.

Michele G. McDevitt
(Type or Printed Name of Person Mailing Correspondence)

Michele G McDevitt
(Signature of Person Mailing Correspondence)

EU647807493US
("Express Mail" Mailing Label Number)

UNITED STATES PATENT APPLICATION

INVENTORS:

Unisys Corporation

APPLICATION:

Computer System Virtualization

ATTORNEY DOCKET NO.

TN333

**Michael Atlass
Attorney for Applicant
Reg. No. 30,606
Telephone No. (215) 986-4111
Unisys Corporation
M.S. E8-114
Unisys Way
Blue Bell, PA 19424-0001**

CERTIFICATE UNDER 37 CFR 1.10: The undersigned hereby certifies that this transmittal letter and the paper of papers, as described hereinabove, are being deposited in the United States Postal Service, "Express Mail Post Office to Addressee" having an Express Mail mailing label number of EU647807493US, in an envelope addressed to: Commissioner for Patents, PO Box 1450 , Alexandria VA. 22313, on this 8th day of October, 2003.

Michele McDevitt 10/8/03

Michele Mcdevitt Date

Computer System Virtualization

Background of the Invention

Field of the Invention:

This invention relates to the field of computer system virtualization. A first implementation provides for virtualization of a plurality of partitions in a multiprocessor computer system.

Background:

Various designs have been proposed for supporting operating systems through use of an OS platform that is separate from the hardware layer to enable optimization of capacity and function in computer systems. We describe in this

Brief Description of the Figures

Various figures are included within this document which comprises the documents referred to in the Detailed Description and which are made a part hereof by this reference thereto.

Detailed Description of the Preferred Embodiments

While various embodiments of the present invention are described, it should be understood that they have been presented by way of example only, and not as a limitation. Thus, the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the Claims that may issue herefrom and their equivalents. The following list of documents which are made a part hereof includes:

1) Supervisor Channels - 14 pages
2) Combined Requirements and Proposed Document – 37 pages
3) Booting EFI on EFI Proces Steps and Work Items – 10 pages
4) (Untitled) Introduction to EFI – 3 pages (EFI Overview)

5) (Untitled) January 24, 2003 Larry Krablin (Food Prep) – 3 pages
6) Monitor: Scan-Fix – 20 pages
7) Supervisor Network Architecture (NetworkDriver.doc) – 4 pages
8) Supervisor EFI.doc (untitled 0116.doc) – 8 pages
9) Supervisor Proposal – 47 pages
10) Supervisor Task List – 17 pages
11) 14 February 2003 (Violation Enumeration) – 13 + 3 pages of intro & TOC – 16 pages total
12) Virtual Disk Driver – 36 pages

Thus, the invention has been described.

ABSTRACT

A soft-channels based architecture for virtualizing partition instances of OS's with loose association to hardware is described herein.

Claims

1. We claim all patentable systems and processes substantially as described within this submission.

Supervisor Channels

By John Landis, Dan Kuklov, Kaike Wan

06/09/2003	Revision 1.0	Dan Kuklov
09/25/2003	Revision 2.0	Kaike Wan

Supervisor Chann ls Overview

Supervisor will use channels (shared memory) as a mechanism for data/command exchange between software partitions. The Memory will be in one Guest partition, but the Host or another Guest will have access to it by mapping the channel into its own memory space. This concept is taken from RDMA mechanism, and supervisor's intent is eventually to follow RDMA to allow hardware acceleration.

The channel will be defined with a generic header that will specify channel memory size, channel type, channel state, and other standard attributes. Standard channel types will provide the infrastructure of the Supervisor.

This document will explain several channel types, their purpose, and their functioning mechanisms.

Channel Endpoints

A channel endpoint is where methods live to interact with a particular type of a channel. Many channels are symmetric – they have the same endpoint methods on both ends, though the end that is listening is normally in a "server" role. A few will have asymmetric endpoints such as the Control Channel. These are special purpose channels.

A server endpoint will include a linked list link for the channel endpoint list and a reference to the channel itself.

Endpoint properties:

Link
Channel

Endpoint methods:

Enqueue
Dequeue

Channel Servers

Channel Servers will implement ChannelServer protocol. This protocol is used for binding a channel to the server. In the initial stage all of the channel servers will be implemented as EFI drivers in the Host partition, but most of them will migrate to their own EFI partitions except the Admin Channel server.

Channel Bus

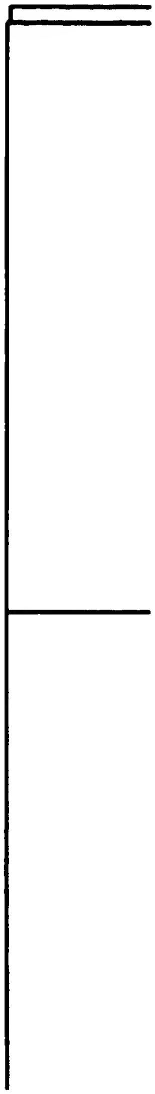
Channel Bus is the interface exposed by the channel mechanism to the clients in the Guest Partition. Its main functions include a method to create a channel for the client, a method to free the channel, and a method for the client to wait for I/O requests to be processed by the Channel Server. Channel Bus communicates with Channel Manager through the Partition Channel and always stays in the Guest Partition.

Channel Manager

Channel Manager is the bridge between Channel Bus and Channel Server during the creation and teardown of a channel. In the initial implementation, it will stay in the Host Partition along with the Channel Servers. Furthermore, it will also mediate the scheduling of the Channel Servers. Eventually it will move into the Admin Partition (or PartitionManager Partition, a Guest Partition), where it will only facilitate the creation and teardown of the channels. Channel Manager has knowledge of all the available Channel Servers and will be instrumental in the channel server assignments for all the Guest Partitions.

Control Chann l

Diagrams



Description

This channel resides on top of the Guest EFI memory and at the bottom of the memory reserved for the Guest OS. It is allocated when a new partition is being created.

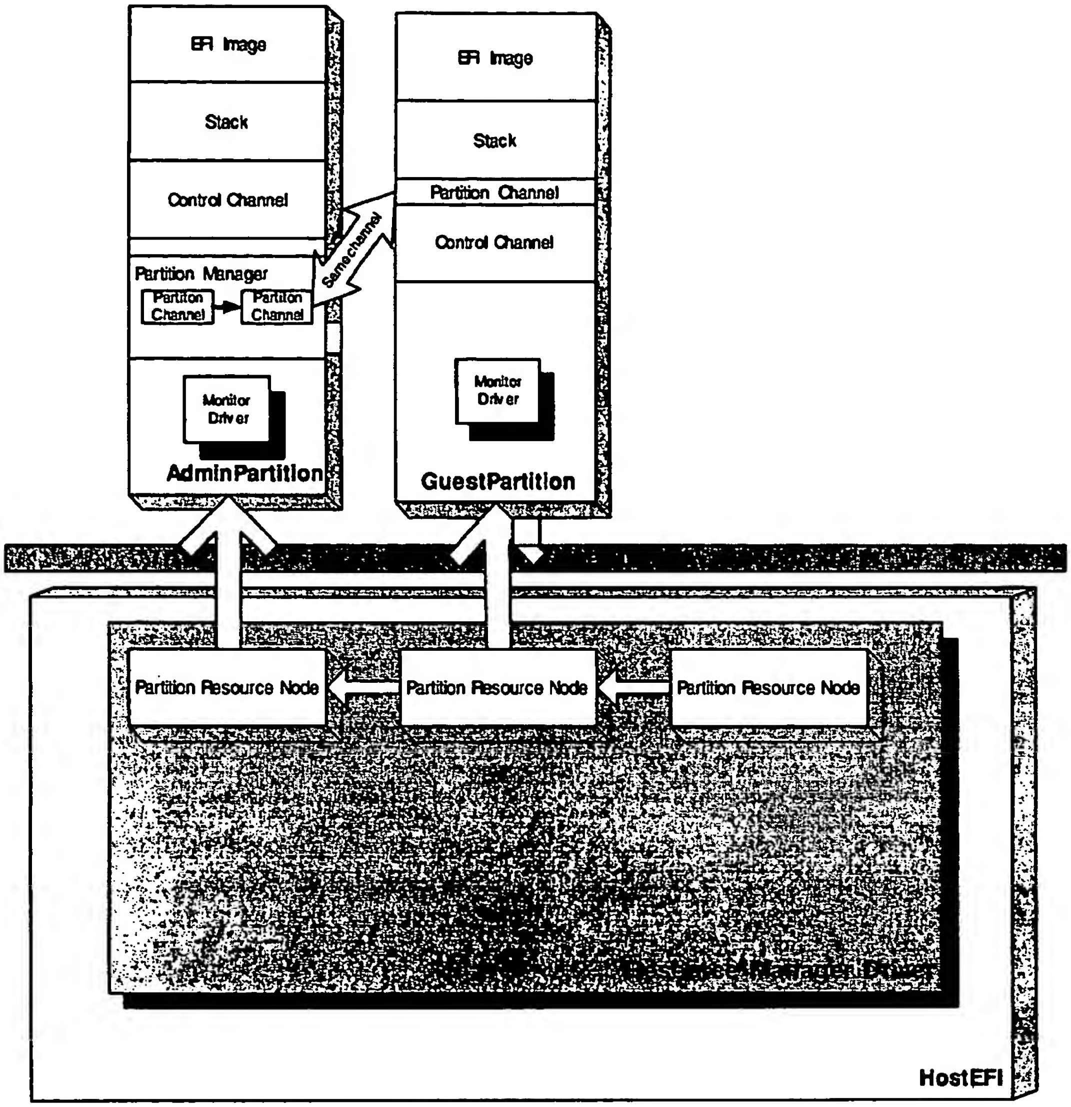
One of the functions for this channel is to deliver hardware interrupts to a partition through the Message Signaled Interrupt mechanism so that the Guest Partitions are shielded from the peculiarity of the underlying interrupt controller. Another function for the Control Channel is to provide status information about the Guest partition to the Resource Manager, such as heartbeat.

Control Channel Manger/Server

The Control Channel Manager will reside in the Host EFI (part of the Resource Manager driver) and will be responsible for dispatching hardware interrupt events (or any other events?) to the appropriate Guest Partition through the corresponding Control Channel. The Control Channel Manager is also collecting status information from the Control Channel and passing it to the Resource Manager for resource management, such as dynamic scheduling based on the need or recovery of a failed partition.

Partition Channel

Diagrams



Description

This channel contains part of the functions of the original Control Channel implemented in the Demo. It provides a communication channel between the Admin Partition and every other Guest Partition and plays an important role in the creation of a new Guest Partition and its virtual devices. During the creation of a new Guest Partition, the Admin Partition allocates memory for the new partition, sets up a Partition Channel within the allocated partition memory, puts all virtual device information (such as virtual disk device) inside the Partition Channel, and finally starts the new Partition. Upon startup, the new Guest Partition will retrieve the virtual device information from the Partition Channel and create the devices accordingly. The Partition Channel also facilitates the creation /deletion of other channels during their attachment to or detachment from the Channel Servers because only Admin Partition has the information for all the available Channel Servers.

Control Channel Manger

The Control Channel manager will be the Partition Manager inside the Admin Partition. It will be responsible for the setup of the Control Channels, and manage the attachment/detachment of newly created channels.

Admin Channel

Diagrams

Description

The Admin Channels, Admin Manager, and Resource Manager will provide the infrastructure to manage partitions in the Supervisor. The presence of the Admin Manager and Admin channels will take all of the configuration logic and the user configuration command parsing from the Host EFI into the Admin Manager. Also, it adds reliability by allowing restarting the Admin Manager if it fails at any point.

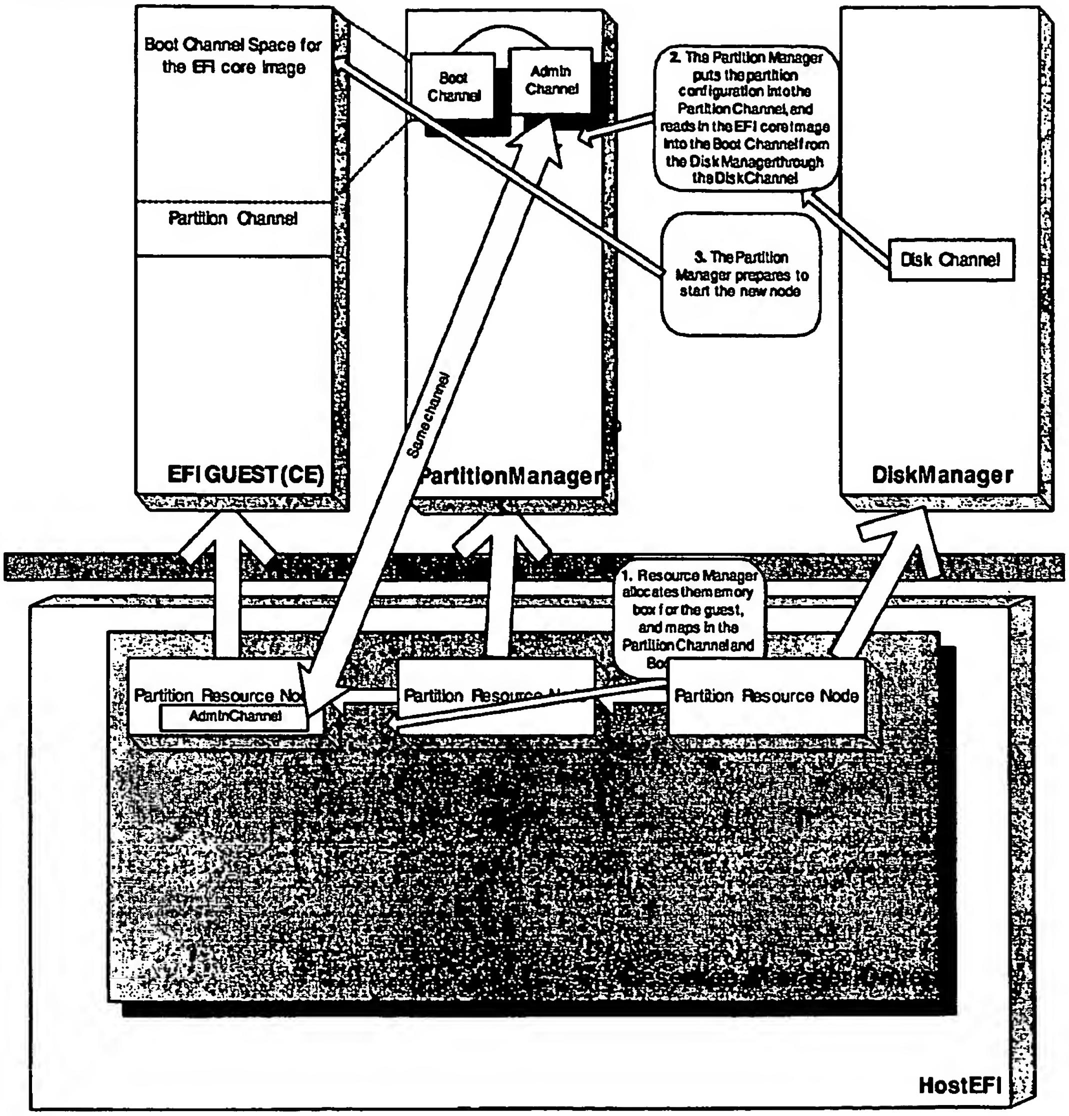
The client channel end point will be the Admin Manager as it sends the configuration change requests or resource assignment requests to the Resource Manager through the Admin Channel. Admin Manager can be hosted by Windows CE guest partition running Compact .NET Framework, which provides a rich platform for system provisioning.

Admin Channel Manager

The Resource Manager is the Admin Channel Manager. It provides the backend for the Admin Manager in the Admin Partition and will be responsible for adding or removing partition resources or saving partition states. The resources can be memory, cpu, storage, network, console, and partition states, which are kept in the Thin Blue Line. At the start of the Admin Partition, the Resource Manager will allocate and set up the Admin Channel before launching it. If the Admin Partition ever fails, the Resource Manager will start another copy of the Admin Partition using the saved partition states in the Thin Blue Line so that the system will run normally without affecting any other Guest partitions.

Boot Channel

Diagrams

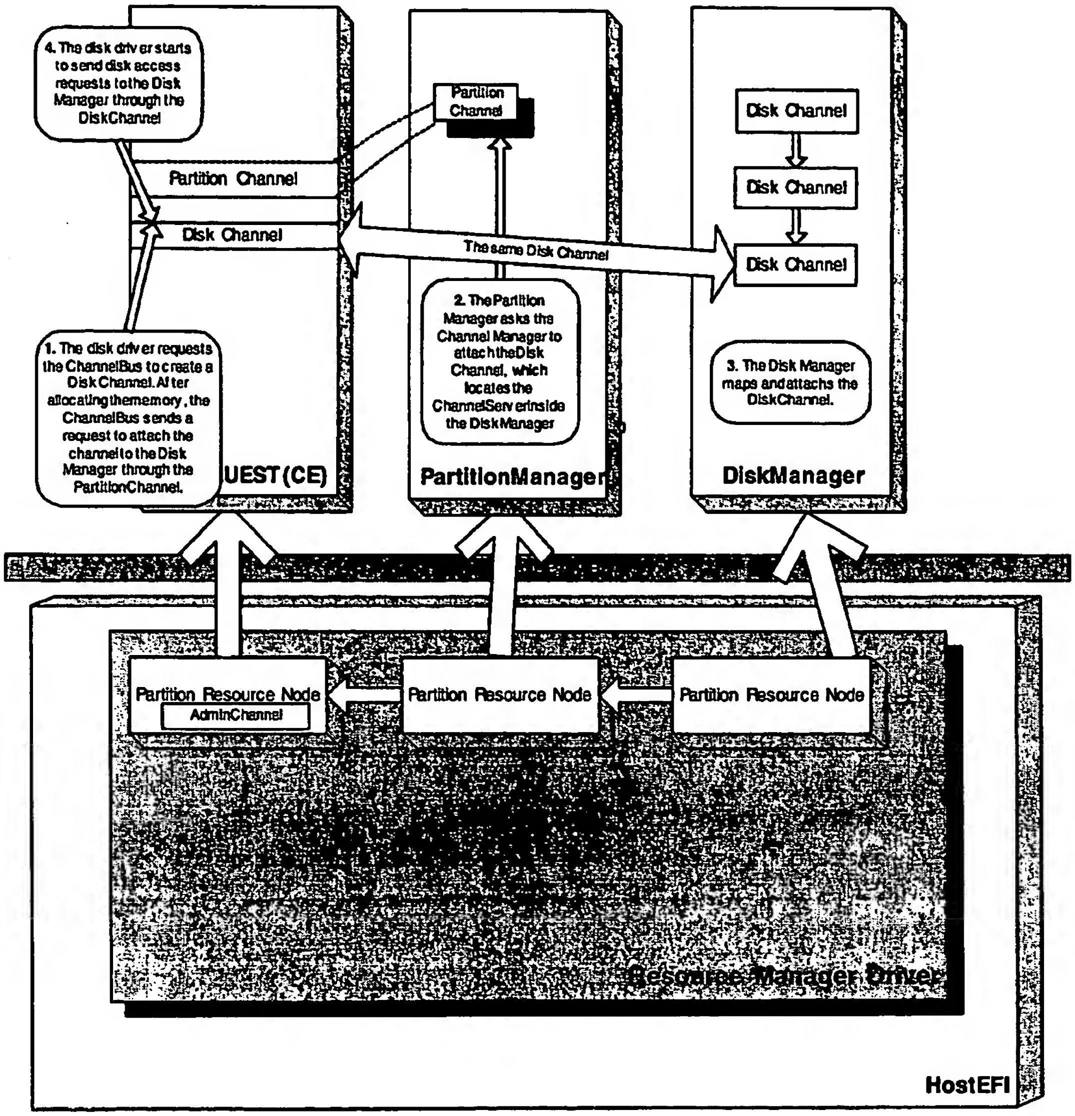

Boot Channel Space for the EFI core image
Partition Channel
EFI GUEST(CE)
Boot Channel
Admin Channel
PartitionManager
2. The Partition Manager puts the partition configuration into the Partition Channel, and reads in the EFI core image into the Boot Channel from the Disk Managerthrough the DiskChannel
3. The Partition Manager prepares to start the new node
Disk Channel
DiskManager
Same channel
1. Resource Manager allocates the memory box for the guest, and maps in the Partition Channel and Bo
Partition Resource No
AdminChannel
Partition Resource N
Partition Resource Node
HostEFI

Description

The Boot Channel provides a mechanism to read in the core firmware image into the partition memory through the Partition Manager, without Resource Manager intervention. The Host EFI will only allocate the memory box for the Guest. Then the Partition Manager will copy the configuration into the Control channel, and read in the EFI firmware. However, in the initial implementation and during the bootstrapping of the Guest Partition containing the Disk Manager in later implementation, the disk I/O will still be performed through the Resource Manager in the Host EFI.

This mechanism allows avoiding most of the I/O processing inside the Resource Manager, and all of the configuration tasks are moved into the Partition Manager.

Disk Channel


4. The disk driver starts to send disk access requests to the Disk Manager through the DiskChannel
Partition Channel
Disk Channel
1. The disk driver requests the ChannelBus to create a DiskChannel. After allocating the memory, the ChannelBus sends a request to attach the channel to the Disk Manager through the PartitionChannel.
UEST (CE)
Partition Channel
The same Disk Channel
2. The Partition Manager asks the Channel Manager to attach the Disk Channel, which locates the ChannelServer inside the DiskManager
PartitionManager
Disk Channel
Disk Channel
Disk Channel
3. The Disk Manager maps and attachs the DiskChannel.
DiskManager
Partition Resource Node
AdminChannel
Partition Resource Node
Partition Resource Node
Resource Manager Driver
HostEFI

Description

The Disk (Storage) channels will provide disk access capability for the Guest Partition in a bid to virtual the disks. There are two types of virtual disks for the Guest Partition: a system volume disk used for booting the Guest OS and non-system disks for the Guest OS. The former disk is presented to the Guest EFI through a virtual disk driver in the Guest EFI during the boot time of the Guest EFI and will contain the Guest OS image and other necessary files such as Guest OS configuration and booting script files. The latter is provided to the Guest OS through a virtual disk driver in the Guest OS instead of in the Guest EFI. In both cases, the client side of the Disk Channel is the virtual disk driver, which calls the ChannelBus protocol implemented in the Monitor driver of the Guest EFI to allocate a disk channel and then attach to the appropriate physical resource under the management of the Disk Manager. The Channel Bus allocates the memory for the Disk Channel, initializes it, and then sends the attach request to the Channel Manager through the Partition Channel. The Partition Manager in the Admin Partition receives the attach request, and forwards it to the Channel Manager, which is also in the Admin Partition. The Channel Manager has a database of all available Channel Servers and calls the appropriate Channel Server implemented by a Disk Manager. Once the Disk Manager receives the attach request, it maps the Disk Channel into its own memory space, allocates necessary physical resources (image file or physical disk), and finishes the initialization of the Disk Channel. From now on, the virtual disk driver and the Disk Manager can communicate with each other through the Disk Channel. For a guest system volume disk, the physical resource for a disk channel is an image file formatted using the FAT file system. For other disks, the physical resource may be an image file or a physical disk partition, depending on the size and performance requirements.

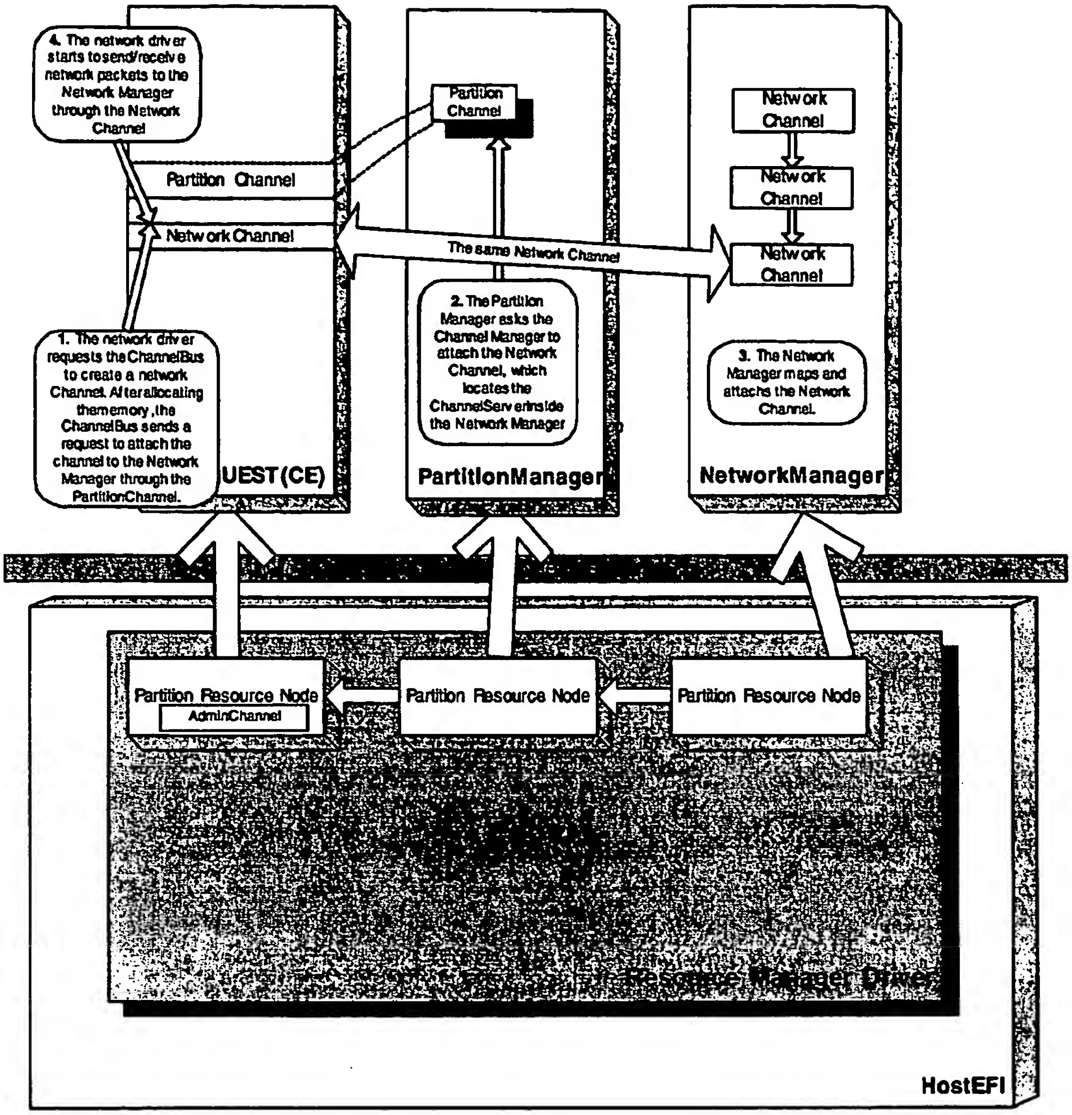
Disk(Storage) Manager

The Disk(Storage) Manager will be the server for disk channels by implementing the ChannelServer protocol. There will always be a copy in the Host EFI for bootstrapping the Admin Partition and the Disk Manager partition(s). However, for bootstrapping other partitions and all runtime guest partition disk services, the Disk Manager(s) in separate partition(s) will be used for better performance and reliability.

Debug Channel

This channel should help with debugging, but there is no implementation plan yet.

Network Channel


4. The network driver starts to send/receive network packets to the Network Manager through the Network Channel
Partition Channel
Network Channel
1. The network driver requests the ChannelBus to create a network Channel. After allocating the memory, the ChannelBus sends a request to attach the channel to the Network Manager through the PartitionChannel.
UEST(CE)
Partition Channel
The same Network Channel
2. The Partition Manager asks the Channel Manager to attach the Network Channel, which locates the ChannelServer inside the Network Manager
PartitionManager
Network Channel
Network Channel
Network Channel
3. The Network Manager maps and attachs the Network Channel.
NetworkManager
Partition Resource Node
AdminChannel
Partition Resource Node
Partition Resource Node
Resource Manager Driver
HostEFI

Description

This channel will send and receive network packets. The client will be a virtual network driver in the Guest OS. The allocation and attachment of a network channel is similar to those of a disk channel.

Network Manager

It is the server for the Network channels by implementing the ChannelServer protocol. In the product it will reside in a separate Guest EFI Partition as a driver.

Consol Chann l

Description

The Console Channel provides a way to debug and start the Guest EFI and Guest OS. After the Guest OS is up and running, other remote client tools such as VNC and Remote Desktop will be harnessed to provide access to the Guest Partition. The Console Channel will allow the Guest EFI to use a console without dependency on the host or any hardware. Its client will be a virtual console driver in the Guest EFI, which exposes the Console Channel as a serial port, essentially providing a text-mode headless console to the Guest Partition. For VGA mode, a UGA protocol interface must be exposed instead

Console Manager

The Console Manager will be the server for the Console Channels. It will switch between consoles and display the characters in the console channel to the screen or transmit the output through the serial ports or potentially through a network. For bootstrapping other system partitions such as the Admin Partition, a copy of the Console Manager in the form of a driver must be present in the host partition. Once the bootstrapping is finished, the Console Manager can be hosted in a separate Guest Partition.

Title:	Supervisor	68948876-002
Type:	Combined Requirements and Proposal Document	Version B
Owner:	Bruce Vessey	
Date:	09/15/03	Page 1 of 38~~23937~~

Document Distribution List

	Name	Location		Name	Location
A	Bruce Vessey	TR C256	O	Terry Powderly	TR A29M
R	John Landis	TR A29F	O	Aravindh Puthiyaparambil	TR E240
R	Eric Benshetler	TR G161-11	O	Andrew Sanderson	TR T244
O	Tim Case	TR B240B	O	Tim Sell	TR C235
O	Michael Salsburg	TR B203H	O	Raj Subrahmanian	TR E223D
O	Derald Caruthers	RV 4863/ 31L21-4	O	Penny Svenkeson	RV 4233 GRID 32N15-1
O	Mo Connell	TR E244	O	Kaike Wan	TR B233
O	Alan Grubb	TR B249	O	Rich Williams	TR E228
O	Jamie Jones	TR K205H	O	Eleanor Witiak	TR E244
O	Jim Hunter	TR C258	O	Joe Zabaga	TR E232
O	Diane Kessler	TR A29B	O	Product Information Document Library	MV 226
O	Renee Korejko	TR F214			
O	Nick Luzeski	TR B244			
O	Brian Parker	TR A29D			
O	Derek Paul	MV 155			

Legend: A - Approver
R - Required Reviewer
O - Optional Reviewer

_______ Approved Without Change

_______ Approved With Indicated Changes

_______ Not Approved

____

Signature of Approver / Date

_______ Comments

_______ No Comments

____

Signature of Reviewer / Date

Comments due by: 10/03/03

Notice : The printed version of this document may not be current.
Verify the version date against the on-line file in the Design Review System DataBase.

Abstract:

This document is a combined Requirements/Proposal for the Supervisor project. Supervisor is a software product that is ~~insalled~~installed on a bare ES7000 partition and provides a flexible repartitioning into many isolated virtual systems.

UNISYS

Supervisor

Combined Requirements and Proposal Document

Date: ~~06/02/03~~09/15/03

68948876-002
Version B

Approval Status: In Review

Name	Role	Location	Mail Stop	Phone
Larry Krablin	Author	TR	B255	NET 385-4397
John Landis	Team Leader	TR	A29F	NET 385-2497
Bruce Vessey	Program Manager	TR	C256	NET 385-4120

This page contains hidden text, which includes the Document Info Table and the macros used to update the table, restore bookmarks in the table, and remove hidden blue text when the document is complete. To view hidden text in Microsoft Windows, turn on the option to view hidden text.

Tabl of Cont nts:

1. Docum nt Control

This document was generated using the PPG Template Generator, 3490 3880, revision H. This document is based on the combination of the PPG Requirements Specification Guide and Template (3486 2391) and the PPG Proposal Document Guide and Template (3486 2409).

NOTE: This document may be reviewed in stages as information is added. For example, one version of this document may contain only requirement information. A later version will also have proposal information. Therefore, earlier versions of the document may contain blank sections if these sections do not pertain to design phase that is being documented. Please refer to the PPG documents listed above to see what information/sections are pertinent to a particular design phase.

1.1. Change History

Revision A is the first version of the document.

Revision B contains comments received and responses to them, plus a small amount of emendation.

1.2. Document Cross Reference

The following are relevant published MSOR documents:

- Dylan 1.0 MSOR-A: file://rv-zeus/data/OPS/PDC/dylan/Dylan-1.0-MSOR-A.doc
- Joplin MSOR A: \\rv-zeus\data\ops\pdc\Joplin\ES7000-Joplin-1.0-Requirements.doc
- Custom Windows App Consolidation MSOR-A: \\rv-zeus\data\ops\pdc\Custom Windows App Consolidation\Custom-Consolidation-MSOR-A.doc

There are a number of project documents, notes, white papers, external references, etc. available through the project web page: http://nioprojects.tr.unisys.com/security/WSP/Supervisor/index.htm.

2. Introduction

2.1. Purpose

This document describes the requirements and proposes the general structure for a software system that will be given complete control of an ES7000 hardware partition (no interposed operating system), and will make the resources of that partition available to client guest software (operating systems and associated applications) as a number of separate, isolated virtual machines.

2.2. Scope

There are a number of requirements for the ES7000 which require the division of its resources into smaller configurations than the physical partitioning mechanisms will allow, in order to provide isolated environments for application systems. The requirements, detailed below, reflect system integrity and security concerns, potential software version conflict issues, as well as customer resource control and billing needs.

The Supervisor product will fill this need by taking control of all the resources of a physical partition (potentially the entire system), and creating virtual partitions that are as isolated, from the client software perspective, as separate physical partitions would be, but not under the same configuration constraints as a physical partition. Each virtual partition will be capable of functioning as a complete IA32 system with its own processor, memory, network connection(s), disk storage and other resources.

2.3. Key Points

Supervisor provides an OS-integrated environment, requiring that low-level ("kernel-touching") guest software (particularly operating systems and associated drivers), be qualified and probably modified to conform to the platform interface.

To the guest software, particularly the guest OS, each virtual partition looks like a separate machine. This may introduce licensing issues for the OS. Of special interest are the licensing requirements for guest software installed into multiple virtual partitions that may not all be active at any given time.

3. Interdependencies

There are requirements that the virtual machines provided by Supervisor be manageable by Server Sentinel in a manner consistent with the management of physical partitions. Interfaces / applets will be provided for this purpose as a part of Supervisor. Work will need to be done in Server Sentinel to invoke them. An agreement needs to be reached as to how and by whom these changes will be made.

There are also requirements, not directly met by the Supervisor product, for automated resource (workload) management across virtual partitions in a hard partition. This may be a requirement on Server Sentinel, which might then have subsidiary requirements on both Supervisor and guest software. This issue is not addressed here.

The *OS-integrated* approach to guest software will require changes to the kernel-touching portions of that guest software. Supervisor will support a derivative of Windows Server 2003 and HAL changes will be necessary. An agreement needs to be reached as to whether these changes will be made by the HAL team in MV or by the Supervisor team.

Similarly, there will be changes necessary to the Windows kernel, requiring cooperation/participation from Microsoft.

It is also possible that BIOS changes may be necessary. If, during design, it's determined that such changes are necessary, then an agreement needs to be reached as to whether these changes will be made by the BIOS team in RV or by the Supervisor team.

4. Overview

4.1. Program Statement

Supervisor will provide more flexible and efficient use of CMP systems by implementing software (virtual) partitioning. To provide better scalability and performance than currently available products, Supervisor will provide an "OS aware" solution. This means that the OS will be modified to eliminate or significantly reduce performance obstacles by working in concert with a firmware-based virtualization layer

4.2. Product Overview

Supervisor is a software product that may be installed in a hard ES7000 partition as the lowest level software. It can then be used to configure and activate a number of virtual partitions within that physical partition. Operating systems, such as Windows Server 2003, suitably modified to use a higher-level abstract interface to the virtual hardware, can then be installed and run in the virtual partition. Non-kernel-touching application software can be hosted, without modification, by the OS in the virtual partition.

5. V ry High L v l R quir ments

There are several broad categories of business requirements for which Supervisor might be thought to provide solutions. These are briefly described here as context for the more formal marketing requirements the project actually responds to.

5.1. Application Consolidation

A major incentive for customers to buy an ES 7000 class system is the ability to consolidate, onto a single system, applications currently being run on many smaller servers. The advantage is lower costs of acquisition and operation, and more flexibility as new applications are added or the workload changes in other ways.

A barrier to this kind of consolidation is that Windows and other operating systems don't isolate applications from interfering with one another, either in terms of resource allocation or system-crash-inducing faults. Additionally, applications often require conflicting versions of support software such as drivers and libraries.

The ES 7000 configuration rules are relatively "coarse", as resources (processors, memory, I/O connectivity, etc.) can only be allocated to partitions in large chunks. This limits the opportunity to use the physical partitioning to get many isolated systems.

Supervisor is seen as addressing this issue by providing, on a single ES 7000 system, a large number of isolated virtual machines, each with resources allocated and managed at a level below the operating system. Each such virtual machine would be an isolated environment tailored to the applications running on it.

5.2. Resource Allocation & Management

When resources such as virtual processors, memory, I/O connectivity and throughput, etc. are seen as "soft", the possibility arises of reconfiguring these fine-grain resources "on the fly" in response to changing application needs. Supervisor may be seen as a vehicle to both facilitate and manage resources in this manner.

5.3. Availability & Fault Isolation

Supervisor will be placed between the hardware platform and any other software, serving to isolate the one from the other. As it may be in complete control of the hardware partition, it may be able to intercept faults of various kinds, preventing them from reaching the software at all or at least significantly reducing the impact they may have on software running in various virtual partitions. There are expectations that Supervisor will significantly improve some measure(s) of overall system availability.

5.4. Platform Design Buffer

There may be an opportunity, as Microsoft places more severe constraints on the HAL, to use Supervisor as a vehicle to repair minor inconsistencies between the hardware and the Microsoft specification.

5.5. Response

Of the several high level requirements mentioned above, the Supervisor project described in this document only really addresses the first: application consolidation.

6. Sp cific Product Requir ments

The specific requirements for Supervisor arise from a number of MSOR documents (see 1.2 Document Cross Reference), as well as some "boilerplate" requirements applying to any product. The items described in this section are the requirements directly responded to by the Supervisor design. Some will be only partially fulfilled in this product. See Appendix B. Requirements Response Summary~~Appendix B. Requirements Response Summary~~ for the complete list and response detail. Specific requirements are linked to this summary by the relevant marketing requirement ids in square brackets.

Requirements called out in this section are given "T" tags that will remain consistent through revision, design, and documentation.

6.1. Terminological Orthodoxy

The existing requirements (cited later) which arise in various scattered MSOR documents and the RequisitePro database have conflicting and/or confusing usages for some key terms applying to the requirements for Supervisor. This section supplies definitions that are consistent and presumably non-controversial. Of particular concern are the terms *virtual* and *physical*, used to describe systems and their components.

A physical resource is one that is, from the Supervisor perspective, provided directly by the underlying system platform. Physical resources and the rules for their inclusion into physical partitions conform explicitly to ES7000 configuration protocols.

A virtual entity is one constructed by the Supervisor software on top of one or more physical resources. From the point of view of guest application software running over Supervisor, virtual resources are indistinguishable from the physical resources they emulate. Kernel-touching guest software (operating systems, etc.) will see differences in the functionality and interfaces for virtual resources. There may not necessarily be any lasting relation between virtual entities and any specific physical resources Supervisor may employ to construct them.

Virtual resources are, in the sequel, always explicitly so identified. An unqualified reference to an entity will always mean a physical resource.

The resources of interest are generally those (processors, memory, etc.) that would be controlled and managed by an operating system.

6.1.1. Guest Software

The term *guest software* refers to notionally off-the-shelf software, including operating systems, device drivers, applications, middleware and other libraries, etc., which would ordinarily perform in the environment of a physical system, using physical resources, but may be made to perform equivalently (more or less transparently) in a virtual environment using virtual resources.

It is important to distinguish between low-level (kernel-touching) guest software, such as operating system kernels and some drivers, and guest applications. The former will be referred to as "guest OS", and the latter will be "guest app".

6.1.1.1. OS-unaware

In an *OS-unaware* environment, kernel-touching guest software (operating systems, device drivers, etc.) need not be modified in any way prior to installation and will operate in the virtual environment transparently.

6.1.1.2.OS-integrated

In contrast, kernel-touching software intended for an *OS-integrated* environment must be modified to use virtual interfaces that are different from (not necessarily direct hardware interaction) the physical environment they were originally designed for.

6.1.2.Virtual Processor

A virtual processor is an abstraction presented by software that appears to guest software, to whatever extent necessary, to act like a real physical processor. The abstraction may be implemented as controlled time slice access to a single or, serially, several single physical processors, or as a full or partial emulation of a processor. In some cases, modifications may be made to the guest software to make it conform to differences between the virtual processor and the physical processor on which it is based. [See discussion of OS-unaware and OS-integrated above.]

The virtual processor presented to guest software will differ from the hard processor it mimics in whatever ways are necessary to constrain that guest software to the isolated virtual partition.

6.1.3.Virtual Memory

There are two aspects of *virtuality* that may be applied to memory.

What appears as a memory address to client software may be mapped to a different address before it is finally resolved to a location in physical memory. There may be several layers of this mapping in use.

Also, at any given moment, a virtual address assigned to client software may refer to a location on a backing medium such as hard disk, rather than an actual RAM location. With this mechanism, it is possible for there to be more virtual memory than physical memory to contain it.

In an OS-integrated environment, the memory allocated to a virtual partition may be one or more ranges of physical memory to which the guest operating system is restricted.

6.1.4.Virtual HBA

A virtual HBA (Host Bus Adapter) appears to the guest software as a physical HBA, with various devices (notably disk storage) behind it and comparable protocols for discovery, usage, and management. The HBA may be implemented by mediating access to a physical HBA and the devices behind it, by an entirely software construction, or by some other communication mechanism to external entities that provide the virtual devices. [M1879.9 , M2066.14]

6.1.5.Virtualized Storage

A virtual storage device appears in the virtual partition to be attached to a virtual HBA. It may be provided by software that allocates and manages access to physical devices and subdivides space on those devices that are attached to a physical HBA in the physical partition, by an interface to external (to the ES7000 physical partition) entity that is accessed through a virtual HBA, or as a completely soft construct using memory or other storage. In any case, capacity, geometry and protocols are defined by configuration and may differ from those of any underlying physical devices. A virtual disk appearing to be attached to a virtual HBA may be treated in all respects as a disk by the guest software, including partitioning, formatting, and installation of file systems.

6.1.6.Network Connections

A physical network connection is represented by a network interface adapter. The adapter has a unique identification (MAC address) and binds one or more IP addresses to the adapter for the physical partition behind it. It serves that partition ("host") in conjunction with the software protocol stack in the host.

A virtual network connection appears to the guest software as an individual adapter, but may be a completely software created interconnection among virtual partitions in a single physical partition, or may be multiplexed with a number of others onto one or more network adapters, with different MAC addresses and associated IP addresses assigned to each of the virtual connections. The virtual connection appears to the guest software to be a physical network adapter.

6.1.7. Virtual Console

On the ES7000 and other systems based on Intel IA32 architecture, an important element of human interaction is the so-called KVM (keyboard/video/mouse) interface. A typical system has at least one of each of these devices either directly attached to it, or, in the case of the ES7000, attached through a console remoting mechanism.

The human interface to a virtual partition presents the same functionality, to both the human and to the guest software, as the physical devices would to a physical partition. The video display appearance and the physical and logical characteristics of the keyboard and mouse are the same as their physical counterparts, but may not necessarily involve KVM hardware directly attached to the physical partition.

6.1.8. Virtual Partition

A virtual partition (virtual machine) is a set of resources sufficient to support the target guest software. The resources may be virtual, such as processors; physical, such as certain memory assigned for the period of active status of the partition; or a virtual resource like one or more HBAs, with whatever virtual storage resources might be attached.

6.2. Hard partition requirements

T1: There are various requirements for the support of hard (physical) partitions.

T1.1: It must be possible to install and run independent copies of Supervisor on any and all hard partitions of an ES7000, subject to size constraints on a hard partition, as noted below.

T1.2: The Supervisor product must be able to manage and utilize all the resources of a hard partition with 64 GB of memory and 32 processors. There are additional requirements (from Joplin) that it be able to support 64 processors and 4 TB memory, but implementation of this has been deferred. [M1879.1 , M2066.5, M2066.6, M1879.1 , M2066.8, M2066.12]

6.3. Independent "virtual machines"

T2: The primary requirement for Supervisor is to be able, through software, to reconfigure a single ES7000 partition into many, fully independent, virtual partitions, each of which is capable of running an appropriate guest operating system and other guest software.

6.3.1. Between silicon and operating system

T2.1: The Supervisor product will interface between the hardware/platform firmware (BIOS) under it and guest software above it. It shall not require any services from an operating system. [M854.6 , M1879.2 , M1879.10 M1879.11 , M1879.12 , M2066.4, M2066.15, M2066.16, M2066.17, M2071]

6.3.2. Supported Operating Systems

T2.2: Supervisor must support Windows Server 2003 and possibly Linux. It must support any combination of guest operating systems in different virtual partitions within the same hard partition. Note that this requirement will be interpreted to allow only properly modified derivatives of these operating systems. [M1879.4 , M2066.7]

6.3.3. Isolation

T2.3: Each virtual partition must be fully independent of others sharing the same physical partition, such that failures of Supervisor or guest software in one virtual partition may not cause failures in another virtual partition. Rebooting a virtual partition shall have no effect on other virtual partitions.

6.3.4. Overhead

T2.4: The virtual partitioning system shall impose a processor overhead of less than 8%. The MSOR items do not indicate any particular benchmark against which to measure this. A suitable benchmark will be specified. [M2066.2]

6.4. Configuration requirements

T3: This section contains requirements relating to the configuration limits supported by Supervisor.

6.4.1. Virtual partitions

T3.1: Supervisor must be able to support, within a single hard partition, more than 500 virtual environments, [Note that it is unclear whether this is intended to mean 500+ simultaneously active virtual partitions, or 500+ simultaneously configured virtual partitions; the former is assumed.] [M854.1 , M2066.10]

6.4.2. Virtual processors

T3.2: There are requirements calling for up to at least 4 virtual processors in a virtual partition, and also (Joplin) for 16 virtual processors in a virtual partition. [M1879.6 , M1879.7 , M2066.9]

6.4.3. Memory

T3.3: Supervisor must be able to provide a virtual partition with up to 4 GB of memory. It must be able to support guest software with as little as 100 MB. [M854.2 , M1879.8 , M2066.13]

6.5. Optional installation

T4: For any physical partition on an ES7000, the installation of Supervisor software and configuration of one or more virtual partitions must be an option for the customer. [M1879.1 , M2066.1, M2066.3]

6.6. Server Sentinel interaction

T5: Several requirements relate to interaction between Supervisor and Server Sentinel.

T5.1: Server Sentinel must be able to monitor and manage the resources of a virtual partition as it does a hard partition. [M854.4 , M1879.13 , M2066.18]

T5.2: Further, Server Sentinel must provide automated resource (workload) management across virtual partitions in a hard partition. [M2066.19]

6.7. Resource Requirements

Specific resource requirements (minima, maxima) are called out and responded to elsewhere.

6.8. Installation Requir ments and Release Packaging

T6: There are requirements for packaging and installation.

T6.1: The Supervisor product must be pre-installable by Unisys prior to shipping an ES7000.

T6.2: The product must also be user re-installable.

6.9. Competitive Requirements

T7: The product must provide clear market advantages over VMWare (GSX Server and ESX Server), Microsoft Virtual Server, and SWsoft Virtuozzo.

Areas to be addressed include:

1. T7.1: better performance than existing products
2. T7.2: availaibility of multi-processor virtual machines

6.10.Standard Requirements

6.10.1.Compatibility

T8: There are requirements for hardware and software compatibility.

6.10.1.1.Hardware Compatibility

T8.1: Supervisor must run on "Rascal" systems. If possible, it is also desirable that it run on "Dylan" systems.

6.10.1.2.Software Compatibility

T8.2: Supervisor must run with OSes and other software (including application software) as described elsewhere in this document.

6.10.3.Documentation

T9: New and updated documentation will be required.

T9.1: A new Supervisor manual and on-line help text will be required.

T9.2: We also expect that the "system capabilities" and "system installation" manuals will require updates. Other documentation may also need updates, and this will be identified as the design progresses.

6.10.6.Internationalization

T10: All user interfaces must be designed to accommodate internationalization.

6.10.9.Performance and Connectivity

Please see section Error! Reference source not found.~~1.1.1~~ Error! Reference source not found.

7. Functional Overvi w

The Supervisor product is a set of software that can be installed by the customer in a physical ES7000 partition. With that software, the customer may establish a number of virtual partitions, each of which may be loaded with a guest operating system and other guest software.

A virtual partition may be activated (powered up) dynamically without affecting other dynamic partitions in the same hardware partition, provided that sufficient physical resources are available to support the configured virtual resources.

A virtual partition may be found in any of various states (see definitions below).

7.1. Virtual Resources

A virtual partition is a collection of virtual resources implemented and managed by Supervisor to provide a complete environment for a guest OS and associated software. Minimum and maximum resource limits are discussed below.

A virtual partition may or may not have guest software installed on it.

The resources assigned to a virtual partition will be constructed/allocated and maintained in various ways, depending on the kind of resource and the state of the partition. Processors and memory are not allocated for an inactive partition.

7.1.1. Processors

A virtual processor will consist of some fraction of the available time of a single physical processor. Supervisor will provide tools to specify the desired performance level of the virtual processor and a scheduling regime to implement the specified performance. For initial release, only one virtual processor will be supported in a virtual partition.

7.1.2. Memory

Physical memory will be allocated to a virtual partition in the amount called for in the configuration for that partition. Supervisor may supply address relocation or may require the guest software to use a specific physical address range designated when the partition is activated. Guest software will be required to use Supervisor-supplied interfaces for all activities involving the allocation and mapping of memory addresses (OS-integrated).The guest operating system is expected to treat the memory address range(s) supplied by Supervisor as a memory address space under its control and may manage that space using a backing medium.

Supervisor will not over commit physical memory, that is, allocate more memory to active virtual partitions than is available in the physical partition.

7.1.3. Disks and other devices

A physical partition may have access to one or more physical HBAs, attached to which may be a variety of devices, physical or virtual (virtual even to Supervisor). Other devices may be connected to the physical partition logically through a network. A virtual partition may be configured to create virtual devices from these resources. Finally, Supervisor may be configured to create virtual devices with no external connection at all (in memory, for instance).

These virtual devices will be presented to the guest software as being attached via a virtual HBA. Persistence of such virtual devices from one period of activation of the virtual partition to the next will depend on the flexibility of the guest software.

Virtual HBAs may be used to connect to shared storage devices, virtual or physical. Use and management of the shared storage is dependent on the storage itself and on the guest software.

7.1.4. Network connections

Supervisor will use whatever physical network interface points are available in the physical partition and provide virtual network interface points (MAC addresses) that appear as distinct interfaces to both the guest software and to entities outside the virtual partition.

Virtual networks that are entirely local to the physical partition will not require any physical network adapter.

Kernel-touching Supervisor software will take care of routing incoming traffic to the appropriate virtual partition. Communication among different virtual partitions within a single physical partition will be transparent.

7.1.5. Console

Supervisor will provide a virtual KVM interface to each virtual partition, where one or more sets of physical keyboard-mouse-video will be connected, not necessarily directly, to the virtual partition. The physical devices may be shared/multiplexed among a number of virtual partitions, by switching, windowing, networking or comparable method. The virtual KVM interface will be transparent to the guest software.

In Windows 2003, the primary operational KVM interface would be the standard Windows Remote Desktop. The Supervisor-provided virtual console will be primarily used for guest software installation and network configuration.

7.2. Virtual Partition States

The virtual partition may be in one of a number of states (nomenclature largely reflects Microsoft Virtual Server usage):

- Uninstalled – The virtual partition has been configured and the configuration recorded, but no resources of any kind have been assigned. The installation process for a partition places it in the stopped state with no guest software installed. A partition that has been installed may be uninstalled, resulting in the release of all resources reserved for it, but with the configuration info retained.
- Stopped – The partition is fully configured, but not actively running. It has no memory resources assigned to it, nor any saved memory image. It will not be scheduled for any processor time. Network messages apparently addressed to it will be dropped. There may or may not be any guest software installed. The *power on* process places the partition in the starting state.
- Starting – The virtual partition has physical memory assigned to it and the virtual processor(s) are being scheduled for processor time. It is in its firmware boot sequence. If an operating system is installed, it will be booted up.
- Running – The virtual partition is running, having completed the hard boot process. If a guest operating system is installed, it has control of the virtual resources. If no guest software is installed, the virtual system firmware is ready for installation.
- Paused – The virtual partition has been interrupted and the virtual processor(s) are not being scheduled for processor time. Memory resources and virtual processors (register contents) remain assigned, with content preserved. Network messages are queued, or may be dropped after some period. The system may be resumed at the point where it was paused.

- Saved – The virtual partition has been paused and its complete state (memory image) has been saved. Its memory resources are released and it is no longer scheduled for processor time. Network message are dropped. The virtual partition may be restored from this state and will resume processing where it was interrupted.
- Active – The partition is starting, running or paused.
- Inactive – The partition is either stopped or saved. An uninstalled partition is not identified as inactive.

7.3. Isolation

Supervisor will manage the virtual partitions such that they are fully isolated from one another. This will be accomplished by modifying the kernel touching code of guest software to invoke Supervisor interfaces for low-level operations, and enforced by forcing the guest software to run above ring 0. Supervisor will maintain separate page tables for each partition, control the interrupt table (fielding many interrupts), and interposing (with the explicit cooperation of the modified guest OS) specialized drivers for all I/O access.

In order to prevent one virtual partition from being affected by failures in another or in Supervisor itself, a "thin" transaction based resource manager will be used to make configuration and management operations atomic, such that, a failure will not leave the system in an a failed state.

Boot up and reboot of a virtual partition are considered normal operations by Supervisor, which will trigger and/or mediate them as necessary.

Within a hard partition under the control of Supervisor, the various virtual partitions may run any combination of the qualified guest operating systems. Each virtual partition will have its own copy of the virtual firmware, the guest OS and whatever other guest software is desired by the customer.

7.4. Management

Supervisor will provide tools for configuration and management of virtual partitions. These will include both direct human interactive tools and interfaces suitable for programmatic management (for Server Sentinel, for example).

7.5. Installation and bootup

The supervisor firmware will be installed in the hard partition in a similar manner to an operating system. As such, installation is optional, for the customer, in any hard partition. Supervisor may be installed in as many hard partitions as desired, running each one separately.

When the hard boot process is started for the hard partition, the supervisor is loaded and takes control of the partition.

7.6. Overhead

When not installed, the Supervisor software will impose no overhead on the system.

When installed but not loaded by the hard partition boot process, overhead will be only for the disk space required for Supervisor installation, any virtual partition configuration information, and any disk space allocated to individual installed virtual partitions, including images of saved virtual partitions.

When installed and loaded, Supervisor will reserve some memory for code and data structures while in operation, both inside each active virtual partition and shared by all virtual partitions.

There will also be some processor overhead, which will be engineered to be minimal. A suitable benchmark will be identified to measure processor overhead.

8. Functional Description

8.1. Components

Supervisor comprises a number of components. These components can be classified by the roles they play in the operation of the system and by their "location" in the system.

8.1.1.Outside Physical Partition

Some software elements run outside the physical partition being managed by Supervisor. Some of these may possibly not even run in the same CEC. They include the configuration, provisioning and management components, which may be run as applications within server sentinel or "standalone" in a separate physical environment, as well as the KVM "clients" that provide the human interface to running virtual partitions.

8.1.2.Inside Physical Partition

Some components reside inside the physical partition being managed by Supervisor.

The "foundation technology" is Intel's Extensible Firmware Interface (EFI), which exists between the platform hardware/firmware and higher level (operating system) software. It provides boot and runtime services and acts as a sort of "sub HAL." It is used by the Supervisor to implement low-level actions requested by the guest OS and to do resource management.

8.1.2.1.Outside Virtual Partition

Some elements are "overhead" resources that are not assigned to any virtual partition, but are required for Supervisor to do its job.

- Supervisor kernel
- Transaction based resource manager – running system state management

Some of the overhead resources that do not run in the customer virtual partitions make use of reserved system virtual partitions.

- LAN services
- HBA services
- Other I/O services – USB, CD-ROM, DVD-ROM, etc
- Provisioning & user interface services
- Disk and lan caches

8.1.2.2.Inside Virtual Partition

In general, Supervisor components within the virtual partition serve to implement functions provided to the guest OS by Supervisor.

8.1.2.2.1.Below OS

The EFI-based monitor platform firmware manages each of the virtual partitions in the physical partition. The monitor enforces containment within the physical resources (processor time, memory space) allocated to the virtual partition. It provides the implementation of the interfaces that the modified guest OS software must use for what would otherwise be kernel-touching operations. It forces the guest OS to run at a protection level above zero and controls the interrupt table, in order to enforce compliance by guest software.

8.1.2.2.2. Within OS

There are a number of Supervisor "intrusions" that appear directly in the guest OS and associated driver stacks:

- modified kernel / HAL
- disk driver
- network driver
- KVM redirection
- other I/O drivers

8.2. Guest Software

Guest software is the standard, notionally off-the-shelf, software that a customer might want to run in a partition, physical or virtual. It includes "guest OS (operating systems, device drivers, ...), applications and libraries. Supervisor provides an OS-integrated environment for guest software. To the extent described below, guest software runs in a virtual partition exactly as it would in a physical partition.

8.2.1. Qualification

All low-level (kernel-touching) software to be installed in a virtual partition must be qualified by Unisys to ensure proper adherence to the low-level interfaces required by Supervisor. These interfaces will be specified and a qualification process defined. Unqualified guest OS software may cause failure of a virtual partition if it touches low-level system state, although the failure would be contained within that virtual partition.

8.2.2. Modification

The qualification of guest OS software will require modification of that software, by Unisys or the software vendor, to meet the interface requirements.

8.2.3. Enforcement

All guest software, in particular guest OS software, will be forced by Supervisor to run at higher protection levels (above ring 0) in order to enforce the exclusive use of Supervisor interfaces for low-level operations.

8.2.4. Supported Guest Operating Systems

A derivative of Windows Server 2003 (not off the shelf -- modified to run over Supervisor) will be supported. Linux may be supported, if required. There is also an opportunity to support Windows CE and run CE-based web servers.

8.3. Configuration Limits

There are various kinds of limit that apply to the current IA32 implementation of Supervisor. These include limits on the configuration of the physical partition under the control of Supervisor, the configuration of virtual components in an individual virtual partition and on the assignment of physical components to a virtual partition.

8.3.1. Physical partition limits

The Supervisor product is intended for use in an IA32 environment. Physical partition configuration limits reflect IA32 architectural constraints, ES7000 configuration rules, and implementation limitations of Supervisor itself. The limits on the physical partition are:

- up to 32 IA32 physical processors
- at least 1 physical processor
- up to 64GB physical memory (present IA32 36-bit addressing limit)
- at least 100MB physical memory
- at least one dedicated network interface card
- at least one dedicated disk unit
- at least TBD MB dedicated disk storage to hold Supervisor software
- sufficient disk space to support all installed (active or inactive) virtual partitions, including configuration information and the contents of memory for a saved virtual partition

8.3.2. Virtual limits for single instance of Supervisor

There are limits to the virtual resources that Supervisor can provide collectively to all the active virtual partitions under its management in a single physical partition:

- Supervisor will support at least 500 active virtual partitions within a single physical partition, physical resources allowing
- the total memory assigned to active virtual partitions is limited to the physical memory available in the partition minus a small amount for overhead
- maximum # procs: TBD
- maximum # of LAN connections: TBD
- maximum # of virtual HBAs: TBD

8.3.3. Limits of virtual partition

Supervisor imposes some limits on the virtual resources that may be configured into an individual virtual partition.

- exactly 1 virtual processor initially, up to 4 in a subsequent delivery
- up to 4 GB memory (virtual always, physical when virtual partition active or paused)
- minimum 100MB memory, subject to guest OS requirements
- at least one virtual network connection
- up to 4 virtual network connections
- up to 4 virtual HBAs

9. Architecture

Figure 1~~Figure 1~~ below shows the basic components and their interrelationships. The scheduling and channel services, along with the BIOS, are the kernel components that reside at the base of the physical partition, above the hardware and below the other elements. All communications among the other components, including the LAN, disk, and user interface services on the left and the individual virtual partitions on the right, are mediated by the kernel.

The monitor component in each virtual partition provides the interface for the guest software into Supervisor for access to the services.

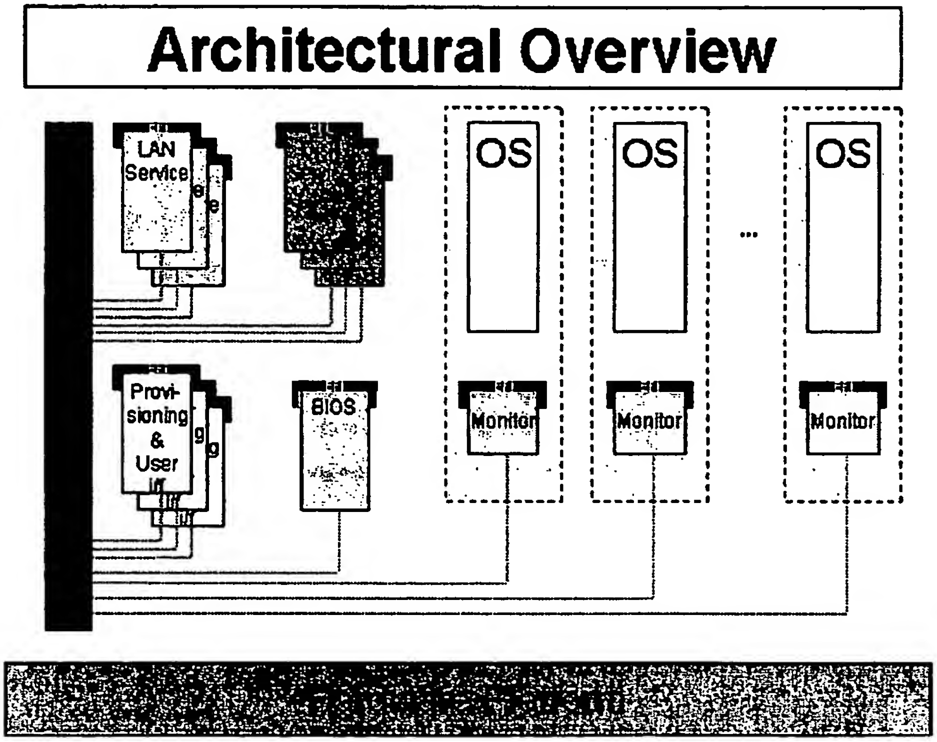


Figure 1: Architectural Overview

Figure 2~~Figure 2~~ shows an individual virtual partition in a little more detail, indicating the local drivers invoked by the guest OS for services provided by Supervisor.

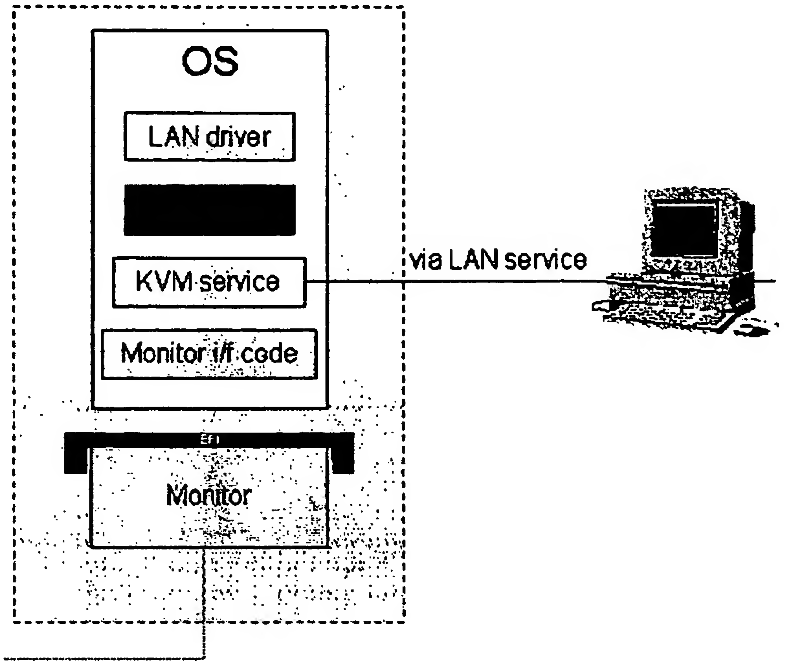


Figure 2: Individual Virtual Partition

10. User Interfaces

User interfaces will be the subject of future designs.

11. System Software Interfaces

System software interfaces will be the subject of future designs.

12. Proposal

12.1.Proposal Summary

See Appendix B. Requirements Response Summary~~Appendix B. Requirements Response Summary~~.

12.2.Compatibility

12.2.1.Hardware Compatibility

Supervisor will run on "Rascal" systems. If possible, it will also run on "Dylan" systems.

12.2.2.Software Compatibility

Supervisor will run with OSes and other software (including application software) as described elsewhere in this document.

12.4.Performance and Connectivity

Please see section 7.6 Overhead

12.9.Internationalization

All user interfaces will be designed to accommodate internationalization.

13. Installation and Release Packaging

The Supervisor product will designed to be pre-installable by Unisys prior to shipping an ES7000.

The product will be user re-installable.

15. Testing

A test plan will be published as a separate document.

17. Issues and Risks

The most significant risk is that the OS-integrated approach requires cooperative changes to the Windows HAL and kernel in order for Windows to operate in the virtual environment.

Appendix A. D finitions, Acronyms, and Abbreviations

See 6.1 Terminological Orthodoxy for definitions/descriptions of a few useful terms.

Appendix B. R quirements R spons Summary

The Requirement Id's used here are those appearing in RequisitePro.

Require-ment Id	Requirement	Source	Section	Commitment	Proposal Response Section
M854	Virtual Server or Virtual Machine	Custom Windows App Consolidation		See sub-items	
M854.1	Set up 500+ Completely isolated "virtual server" or "virtual machine" (VM) environments within the server	Custom Windows App Consolidation	6.4.1	Yes	8.3.2
M854.2	100MB Memory for each Virtual environment	Custom Windows App Consolidation	6.4.3	Yes	8.3.3
M854.3	Allocate I/O to each Virtual environment (minimum ability to allocate 1 PCI slot to a VM).	Custom Windows App Consolidation		No (possibly in conflict with M854.1)	
M854.4	Manage entire 'Virtual environment" with Sentinel	Custom Windows App Consolidation	6.6	Yes	7.4
M854.5	Clustering and Failover	Custom Windows App Consolidation		Not directly addressed here, but intended to work. It will be thought about more as the design proceeds, but the bulk of the requirement falls on the guest OS. There will be consideration of cases where the OS relies on specific underlying hardware.	
M854.6	Isolation: No virtual environment may bring down another virtual environment instance	Custom Windows App Consolidation	6.3.1	Yes	7.3
M854.7	Provide rapid/automatic upgrade/deployment of software updates to all virtual environments. (eg. new versions of virus software being applied)	Custom Windows App Consolidation		Not addressed here – dependent on Server Sentinel and guest operating system, applications	

Requirement Id	Requirement	Source	Section	Commitment	Proposal Response Section
M854.8	Resource Allocation Prioritization: If more than one VM shares I/O, Memory or Processors, a priorization mechanism must exist to determine which application (VM) will get access to the resource in a crunch or peak period.	Custom Windows App Consolidation		Not addressed here – dependent on Server Sentinel and guest operating system, applications. Supervisor will provide relevant interfaces to Server Sentinel	7.4
M854.9	Threshold handling: In manual mode, if a VM begins to hit a threshold, the server will provide the administrator with recommendations as to how to best re-allocate resources based on what is 'typically' underutilized at the specific time the need occurs.	Custom Windows App Consolidation		Not addressed here – dependent on Server Sentinel and guest operating system, applications. Supervisor will provide relevant interfaces to Server Sentinel	7.4
M854.11	Each Virtual Server or Virtual Machine should not require a separate instance of the OS.	Custom Windows App Consolidation		No	2.3
M1879	ES7000 "Dylan" must provide virtual partitioning for resource management	Dylan		see sub items	
M1879.1	Use of virtual partitioning must be optional for the customer.	Dylan	6.5	Yes	7.5
M1879.2	The virtual partition mechanism must operate below the operating system level; i.e. between the hardware and the operating system(s)	Dylan	6.3.1	Yes	8.1.2
M1879.3	Each virtual partition must be capable of running its own operating system.	Dylan	6.2	Yes	7.3
M1879.4	Virtual partitions must be capable of supporting a variety of operating system environments including, but not limited to, Windows 2000 AS and DCS, Windows .NET Web Server, Server, EE and DCS, UnixWare 7.1.1, Open Unix 8.x, Linux	Dylan	6.3.2	Partial: the supported operating systems will be a derivative of Windows Server 2003 and, if required, Linux.	8.2.3

Requirement Id	Requirement	Source	Section	Commitment	Proposal Response Section
M1879.5	Virtual partitioning must run on at least 8 processors; i.e. one Dylan cell.	Dylan	6.2	Yes	8.3.1
M1879.6	A virtual partition shall have a maximum size of at least 4 processors.	Dylan	6.4.2	On initial release, a virtual partition will be limited to 1 virtual processor. A later release will provide up to 4 virtual processors.	8.3.3
M1879.7	Virtual partitioning must be able to utilize at least 16 GB of memory (for all virtual partitions.)	Dylan	6.4.2	Yes	8.3.1
M1879.8	A virtual partition must be able to address at least 4 GB memory but shall not require 4 GB memory.	Dylan	6.4.3	A virtual partition will be able to address 4 GB memory, but there will be some overhead for Supervisor.	8.3.3
M1879.9	Virtual partitions must be capable of sharing I/O adapters in the same cell and other cells that are in the same hard partition	Dylan	6.1.4	Yes	7.1.3,7.1.4
M1879.10	A failure in one virtual partition must not impact any other virtual partition	Dylan	6.3.1	Yes	7.3
M1879.11	Each virtual partition environment must be isolated allowing a reboot of one virtual partition without affecting other virtual partitions.	Dylan	6.3.1	Yes	7.3
M1879.12	The virtual partition mechanism must not introduce additional availability issues. In particular, the virtual partition mechanism can never cause a virtual partition failure.	Dylan	6.3.1	Yes, to the extent practicable. Never say never.	7.3
M1879.13	Server Sentinel must be able to monitor and manage the resources of a virtual partition as it does a hard partition.	Dylan	6.6	Supervisor will provide resource management tools to Server Sentinel comparable to those for hard partitions.	7.4

Requirement Id	Requirement	Source	Section	Commitment	Proposal Response Section
M1879.14	Server Sentinel must provide automated resource (workload) management across virtual partitions in a hard partition.	Dylan	3	Not addressed here – dependent on Server Sentinel and guest operating system, applications. Supervisor will provide relevant interfaces to Server Sentinel	7.4
M1954	Platform specific resource virtualization must allow for running multiple operating system instances in the same physical partition for application isolation	Dylan		Yes	7,0
M2054	Virtual partitioning for workload management and improved application availability.	Joplin		Yes: fundamental motivation for Supervisor. Available only for IA32.	
M2066	ES7000 Joplin must provide virtual partitioning for resource management	Joplin		see sub items	
M2066.1	Use of virtual partitioning must be optional for the customer.	Joplin	6.5	Yes	7.5
M2066.2	The virtual partitioning mechanism must not consume more than 8% of the processor resources when invoked.	Joplin	Error! Reference source not found.~~1.1.1~~	Yes	7.6
M2066.3	If not implemented, the virtual partitioning mechanism shall not consume server resources.	Joplin	6.5	Yes: Supervisor consumes no resources in a physical partition if not installed.	7.6
M2066.4	The virtual partition mechanism must operate below the operating system level; i.e. between the hardware and the operating system(s)	Joplin	6.3.1	Yes	8.1.2
M2066.5	Each virtual partition must be capable of running its own operating system.	Joplin	6.2	Yes	7.3
M2066.6	Each hard partition must be capable of running its own virtual partition environment.	Joplin	6.2	Yes	7.5

Requirement Id	Requirement	Source	Section	Commitment	Proposal Response Section
M2066.7	Virtual partitions must be capable of supporting a variety of operating system environments including, but not limited to, the current versions of Microsoft Windows operating environments equivalent to .NET Web Server, Server, EE and DCS, and Linux.	Joplin	6.3.2	Partial: the supported operating systems will be a derivative of Windows Server 2003 and, if required, Linux.	8.2.3
M2066.8	Virtual partitioning must run on at least 64 processors; i.e. one Joplin cell.	Joplin	6.2	IA32 implementation is limited to 32 physical processors.	8.3.1
M2066.9	A virtual partition shall have a maximum size of at least 16 processors.	Joplin	6.4.2	On initial release, a virtual partition will be limited to 1 virtual processor. A later release will provide up to 4 virtual processors.	8.3.3
M2066.10	Joplin servers shall support at least 500 completely isolated virtual partitions.	Joplin	6.4.1	Yes	8.3.2
M2066.11	Virtual partitions shall be capable of being clustered to the extent of the operating environment utilized.	Joplin		Not directly addressed here, but intended to work. It will be thought about more as the design proceeds, but the bulk of the requirement falls on the guest OS. There will be consideration of cases where the OS relies on specific underlying hardware.	
M2066.12	The total virtual partitioning mechanism must be able to utilize at least 2 TB of memory.	Joplin	6.2	The IA32 implementation of Supervisor is initially limited to 64 GB.	
M2066.13	Each virtual partition must be able to address at least 4 GB memory but shall be able to operate with 100 MB addressable memory.	Joplin	6.4.3	Yes	8.3.3
M2066.14	Virtual partitions must be capable of sharing I/O adapters in the same cell and other cells that are in the same hard partition.	Joplin	6.1.4	Yes	7.1.3,7.1.4

Requirement Id	Requirement	Source	Section	Commitment	Proposal Response Section
M2066.15	A failure in one virtual partition must not impact any other virtual partition	Joplin	6.3.1	Yes	7.3
M2066.16	Each virtual partition environment must be isolated allowing a reboot of one virtual partition without affecting other virtual partitions.	Joplin	6.3.1	Yes	7.3
M2066.17	The virtual partition mechanism must not introduce additional availability issues. In particular, the virtual partition mechanism can never cause a virtual partition, hard partition or server failure.	Joplin	6.3.1	Yes, to the extent practicable. Never say never.	7.3
M2066.18	Server Sentinel must be able to monitor and manage the resources of a virtual partition as it does a hard partition.	Joplin	6.6	Supervisor will provide resource management tools to Server Sentinel comparable to those for hard partitions.	7.4
M2066.19	Server Sentinel must provide automated resource (workload) management across virtual partitions in a hard partition.	Joplin	6.6	Supervisor will provide resource management tools to Server Sentinel comparable to those for hard partitions.	7.4
M2071	Virtual partitioning capabilities must permit application isolation such that the failure of the application will not impact applications in other virtual partitions.		6.3.1	Yes	7.4

App ndix C. Alt rnativ s Consid red

The primary alternative considered was the OS-unaware approach where the guest software is supported in the virtual partition with no modification at all. This entails trapping attempts by the guest to perform certain low-level operations (modifying the interrupt table, for instance) that would interfere with Supervisor's ability to manage the virtual partition and keep it isolated from other virtual partitions. The OS-unaware approach is technically more difficult and is likely to have a fairly severe performance penalty. There do already exist products for the ES7000 in this space (notably VMWare), and achieving market position in this segment is extremely unlikely.

The OS-integrated approach offers the opportunity for better performance and market differentiation.

Appendix D. Comments & Responses from Revision A

From Daniel Ottey:

Section 5.3.2

The list of supported guest operating systems (Windows Server 2003, possibly Linux) does not included Windows NT 4.0. I understand that there would be complexities in incorporating Windows NT 4.0 in a OS-aware approach. My concern, however, is that Microsoft (with their Virtual Server) seems very inclined to use it to help migrate current users of Windows NT 4.0 onto newer hardware (their intent being to have them purchase a copy of Windows Server 2003 at the same time). And it sounds to me like they still have a large customer base who would want to do this. If Supervisor does not support Windows NT 4.0, I think Unisys will miss out on a lot of Supervisor-customers to MS Virtual Server. I believe the number of applications that will be running under 1x-4x instances of Windows Server 2003 that actually require isolation is not going to be very high. Unisys/Supervisor should seek to migrate/preserve current Windows NT 4.0 environments, thus giving us a larger potential customer base.

Understood. However, support for Windows NT appears in no Business or Marketing requirements (see M1879.4 , M2066.7), and, given the OS-aware approach, causes a great deal more work. On the other hand, the technical approach does not rule out later inclusion of NT 4.0.

Section 5.4.1 and 7.3.2

A 500 virtual machine requirement seems very pointless - assuming you want all 500 virtual machines to be running at the same time. Even in a 32x environment, this allows only 0.064 processors per virtual machine (or 15.625 VMs/proc). The performance achieved from this scenario would just be terrible.

The market for that kind of requirement is seen as webservers, competing against products like Virtual Linux. Given the consolidation requirement of separation, the relatively low performance of individual virtual partitions should be acceptable for individual isolated servers.

App B. M854.5 and M2066.10 Clustering and Failover

You say "but the bulk of the requirement falls on the guest OS." In our experience with both VMware GSX and Microsoft Virtual Server, the issue of clustering always involved the virtualization software's support for sharing (virtual) storage between virtual machines. Of course the guest OS (Windows) knows how to cluster machines given shared storage for quorum between the machines. But the virtualization software had to allow two (or more) virtual machine so share the same storage. Does/Will Supervisor provide the ability to share storage between two or more virtual machines?

Clustering depends on shared storage, although Microsoft's Majority Node Set does not require even that. Supervisor will provide access to shared storage resources (directly attached SCSI and/or FC) in a manner consistent with the requirements of [OS] cluster services in a native, not virtual, environment.

App B. M854.3

I understand that there won't/shouldn't be a direct assignment of a PCI slot to a virtual machine. But, does Supervisor have the ability to access external storage via the PCI bus - to Unisys Storage Sentinel and other such storage devices that require the use of fiber-optic HBAs? I believe that it is important to be able to provide virtual machines with a large amount of storage.

Yes, that is the intent behind the discussion of virtual HBAs (6.1.4).

App B. M2066.17

The commitment to this requirement should probably match the commitment to requirement M1879.12

Corrected: the commitments now read the same.

From Derek Paul:

I was surprised to see that the approach you appear to be taking will require modification to the Guest OS and/or the HAL.

As we investigated possible directions, we realized that we really had two options. (1) Do something that does not require any OS changes. I.e. emulate all the underlying h/w just like VMware and Virtual Server do. In this case, we wouldn't have any better performance and, let's face it, they're already years ahead of us. So, we could spend a lot of development effort and come out with something that's almost as good as what's already there. That doesn't sound like a winning proposition. (2) Do something completely different. We think the best bet is something along the lines of what we're proposing.

Is this something we already have agreement on with Microsoft? While I do not have the same degree of interaction with them as you do, my observation had been that they do not appear to approve of this kind of activity taking place. From a business standpoint, it's also not clear to me why they would modify the OS to support our VM type product when they have a similar one of their own.

We do not have an agreement. Over the past year and a half (or longer?) that we've been discussing virtualization with them, they've kept stating that they'd like to pursue an approach where there's support in the platform -- and they explicitly called it "firmware support" -- for virtualization. We believe that what we're proposing is in line with their stated direction. We could have just gone to them with powerpoint-ware, but we thought it best to build a working prototype. We have that almost complete. At this point we need to complete the prototype, protect our intellectual property, and make our sales pitch to Microsoft. If they buy it, great. If not, then we could do a Linux version (if management decides they want to go in that direction), go back to the drawing boards and develop another "me too" project, or cancel Supervisor all together.

In addition, I thought one of the original requirements for this when it was first discussed a year or so ago was to provide the ability for us to 'fix' our HW. The bulk of the changes in the HAL are to work around HW issues discovered during testing of our systems, not for the different architecture of our platform. Microsoft wants to get people out of the HAL business, as we have seen with IA64. I've also seen the results of this, with our inability to provide timely fixes for platform problems. We almost didn't make the original IA64 release because of this. While it may not currently be a Supervisor requirement for partitioning, I would suggest that we should be thinking along the lines of providing an 'ASIC fix' capability to protect ourselves and our ability to ship systems should Microsoft not supply HAL kits on future OS's. (Granted, based on the initial point, that your implementation will require modifications to the HAL/kernel, this whole thing may be a moot point if there are no HAL kit's in the future.)

> We don't want to own/modify the HAL. What we want to do is work with Microsoft such that they own & support a "virtualized" HAL. By this we mean the following. The OS and HAL would be modified such that they expect to run on top of a virtualization layer instead of running on top of actual hardware. Or viewed slightly differently, they would expect to run on a virtualized model of an underlying platform. In that case, Supervisor would be the only software interacting directly with the underlying h/w platform. This should give us an ability to 'fix' some aspects of our h/w. On the other hand, we don't expect this to be a panacea for all possible platform problems.

While it is an implementation detail, in Section 6.3 it's stated that isolation between virtual machines will be enforced by forcing the guest OS to run above ring 0. When I had looked into this a year or so ago, I had come to the conclusion that this was not sufficient to ensure isolation and correct behavior of a guest OS. The X86 architecture is not fully 'virtualizable' in hardware, so there had to be a mechanism to hunt down and virtualize the offending instructions.

> It is correct that some instructions (sgdt, sidt, ...) prevent fully virtualizing a guest OS simply by running it above ring 0, since these instructions allow non-ring 0 code to see privileged state, and others (like popf) would not exhibit the necessary behaviors. However, this exposed state is a liability only to the correct behavior of the current guest, rather than a liability of interference with other virtual partitions. In practice most of these instructions are used very seldom. A few, like popf are more prevalent.
>
> Preventing the OS from running at ring 0 is sufficient (given necessary safeguards) to prevent it from affecting other virtual partitions, which is the primary goal of isolation. So maybe a definition of 'Isolation' would be in order. Our primary goal is to fully isolate virtual partitions from interference by other virtual partitions. (Otherwise scalability is a myth.)
>
> Isolating individual virtual partitions from knowledge of life in a virtual partition is of much less concern (and 'costly' on known x86.) So extraordinary measures to prevent useless usage by the Guest OS software of visible protected state is a non-goal.
>
> Since there is nothing a guest OS (above ring 0) can do with this information except confuse itself and/or crash, ignoring the leakage and instead focusing on cooperating with the Guest OS to provide mechanisms to optimize performance within virtual environment provides more value.
>
> Requiring minor changes to a guest OS to avoid tripping over this leaked state removes the need for supervisor to hunt down the evil instructions. Additional changes to a guest OS to minimize protection exceptions are also useful to minimize performance overhead.
>
> 'Porting' a guest OS to this virtual-x86 mostly involves interrupt and memory issues. NT kemel memory management code does not call HAL since of course it assumes the virtual memory resources are part of the architecture.

Also, does VMware not have patents on some of the technology in this area. Are we aware of them and do we know we can work around them?

> VMware does hold some patents in this area. Connectix also held patents in this area; and VMware & Connectix were battling in the courts. We don't know where that stands now; presumably Microsoft owns the Connectix patents. We believe that that approach we're taking, by modifying the OS as opposed to emulating the underlying hardware, keeps us out of trouble with the VMware & Connectix patents. Obviously if we can strike a deal with Microsoft, that would help.

From Michael Salsburg:

Section 5.4.4 [now Error! Reference source not found.~~1.1.1~~] – The commitment to low processor overhead should be made in the context of a known workload. Therefore, it is suggested that the performance goal be set to not exceed 8% of CPU utilization when running a TPC-C workload.

> The requirement has been amended to specify an eventual choice of a suitable benchmark.

Section 5.6 [now 6.6] – It is incorrect to assume that Server Sentinel will provide automated resource (workload) management across virtual partitions. That is outside the functional realm of Server Sentinel. Workload management may be implemented in a number of places, including Supervisor itself, or one of the modules of "Application Sentinel".

> This is a specific marketing requirement being responded to. The technical design may further delegate the responsibility.

Appendix A – Hierarchical Requirements

> [Note that these comments discuss "independent applications" and the interactions among them. Supervisor does not recognize applications per se, but rather creates virtual environments, each of which may host one or more applications, at customer direction.]

1 High Availability - Significantly improve the availability of Windows systems over and above what can be accomplished with "off the shelf" products

1.1 Reduce hard stops that originate as hardware errors before they induce operating system stops or other hardware timeouts

1.2 Reduce the time to recover once an error occurs

> These items fall under the High Level Requirement "Availability & Fault Isolation (5.3) and are not specifically addressed by this proposal. See 5.5 Response.

2 Virtualization - Provide an environment where applications can be executed and maintained that is not dependent on the underlying infrastructure or other applications

2.1 Many applications shall share infrastructure resources but administered as if they were running alone on dedicated resources

2.1.1 All infrastructure resources shall be made available, including all 32 processors and 64 GB of an ES7000 class system

2.1.2 Both Windows Server 2003 and LINUX operating systems will be supported

2.1.3 ES7000 series, starting with "Dylan", must be supported

2.1.4 Within a "hard partition", at least 500 virtual partitions will be available for independent applications

2.1.5 Each virtual partition shall provide up to 4 GB of memory and shall support guest software that uses as little as 100 MB of memory

2.2 One application's failure shall not affect the availability of another independent application

2.3 One application's resource consumption shall not cause another application's performance to degrade

> These are fundamental requirements for Supervisor and are called out and responded to in this document.

2.4 One application may require multiple distributed resources but shall be administered as if it is executing on a single system

Supervisor does not address this issue beyond the capabilities that may be provided by the guest OS.

2.5 Underlying resources supporting applications shall be able to be changed without application interruption

This is not presently a marketing requirement, and its accomplishment is very dependent on guest OS capabilities.

2.6 The cost for administration and maintenance of multiple virtual operating systems shall not exceed the cost of administering and maintaining each independent application using its own resources

2.6.1 Centralized management, which integrates with Server Sentinel, shall be provided

2.6.2 The virtualization software must be pre-installable by Unisys prior to shipping the ES7000. It must also be user-re-installable

These are fundamental requirements for Supervisor and are called out and responded to in this document.

Appendix D. Business Requirements

Michael Salsburg forwarded a table of business requirements that relate generally to Supervisor. These are listed here as background. They are reflected in the marketing requirements cited in Appendix B. Requirements Response Summary~~Appendix B. Requirements Response Summary~~.

Requirements	Program / Technology [1 - Flt:Y Srt:A]	Reqmt Owner
B20: 25% Lower TCO than competition for equal workload	Consolidation	Ziegler, S
B21: Guarantee Application SLO	Consolidation	Ziegler, S
B21.1: User specifies business apps	Consolidation	Ziegler, S
B21.2: User specifies app's service level	Consolidation	Ziegler, S
B21.3: User sets SLO	Consolidation	Ziegler, S
B21.4: App's service will not be impacted by other apps	Consolidation	Ziegler, S
B22: Chargeback	Consolidation	Ziegler, S
B22.1: User specifies chargeback accounts	Consolidation	Ziegler, S
B22.2: Resource utilization is charged back to accounts	Consolidation	Ziegler, S
B23: Availability	Consolidation	Ziegler, S
B23.1: Total User-Experienced Service Level Availability is 99.999% "out of the box"- without customer heroic effort and without a premium (e.g., the cost of clustering)	Consolidation	Ziegler, S
B23.2: One application failure will not impact other apps	Consolidation	Ziegler, S
B24: Simplify Management	Consolidation	Ziegler, S
B24.1: This includes Solution/Application/System	Consolidation	Ziegler, S
B24.2: Reduce management time by 50%/year	Consolidation	Ziegler, S

Requirements	**Program / Techn l gy [1 - Flt:Y Srt:A]**	**R qmt Owner**
B25: Consolidation Solution Delivery	Consolidation	Ziegler, S
B25.1: Reduce elapsed time by 50%/year	Consolidation	Ziegler, S
B25.2: Solution sizing capability that ensures optimum configuration	Consolidation	Ziegler, S
B26: Investment protection	Consolidation	Ziegler, S
B26.1: Field Upgradeable CPU/Memory/Storage (everything!) Technology	Consolidation	Ziegler, S
B26.2: Uninterrupted Upgrade	Consolidation	Ziegler, S

Booting EFI on EFI process steps and work items.

Introduction

This document is meant to familiarize the reader with the process that takes place to start an EFI Guest instance inside a running EFI Host.

Definitions

There are several names and terms that are used throughout this document:

- ***EFI Host*** – is an EFI core firmware that runs in paged mode with large pages (in our case this will be the visor32 build (ia32-emb build with paging enabled)).
- ***EFI Guest*** – is an EFI core firmware that was modified to run on top of the EFI Host. This EFI will eventually run in paged mode with large pages. The modifications to this EFI include drivers that use the drivers of the EFI Host to abstract devices. The EFI Guest binary image is constructed from several distinct images. There is the EfiLoader what is a small piece code that will bootstrap the EFI guest core firmware. Then there is the compressed image of the EFI guest core firmware. Finally this can be followed by a number of compressed driver images.
- ***Efi32Switcher*** – or simply Switcher, is a driver that is loaded by the EFI Host. The Switcher manages creation of the EFI Guests and switching and scheduling between Host and EFI Guests.
- ***EfiLoader*** – is a module that is a part of the EFI Guest image. This module is responsible for setting up the EFI Guest running environment and unzipping and relocating EFI Guest core firmware.
- ***EFI Guest core firmware*** – is an EFI core firmware that will initialize the EFI environment.
- ***Other EFI drivers*** – some drivers can be compressed and concatenated after EFI Guest core firmware image in the EFI guest binary. These drivers will be uncompressed and loaded by the EFI core firmware during its initialization.

EFI on EFI bootstrap process

The process of loading an EFI Guest on top of the EFI Host can be described in several steps. First, the Switcher creates the structure to manage the EFI Guest, allocates memory, and loads the EFI Guest binary image into memory. Then the EfilLoader relocates itself to actual load address, then copies and relocates to high end inside that memory and calls itself again. The new pass in the EfiLoader will create the running environment for the EFI Guest – this includes GDT, IDT, PD and EFI memory descriptors. Then it will uncompress EFI Guest core firmware and jump to it. The EFI

Guest core firmware initialization routine will finish the creation of the EFI environment by setting up the System Table, and loading all of the needed drivers.

Creation of the Switcher environment for the EFI Guest

The switcher will have a list of structures to manage all of the EFIs that it schedules. The structure will contain the pointer to the memory region for this EFI instance and a current ESP pointer for this EFI.

Steps in the Switcher:

1. The Switcher will allocate large memory pages for the EFI Guest
2. The Switcher will load the binary image of the EFI guest into the 0x0 offset of the last large page in the allocated memory region.
3. The Switcher will set up a temporary 32k stack on (EFI image size + 0x8000).
4. The Switcher will set up the stack for the call into EfiLoader. This includes creating the handoff structure that will contain an EFI memory descriptor, binary image information, and a pointer to the EFI Host System Table.
5. The Switcher changes its ESP and jumps into the EfiLoader.

Diagrams:

EFI Guest binary image consists of several linked parts:

- Small uncompressed EfiLoader, which is the entry point into the EFI Guest. EfiLoader also contains efiselfreloc.asm which intercepts the entry address to relocate the image to the actual load address before transferring to the existing C entry point. The existing C code decompresses the Core.Z image.
- Core.Z is a compressed image of the EFI core firmware. It also contains VmState.asm that intercepts the image entry address and will have the GDT/IDT/PD for the new VM.
- Diver1.z – Driver2.z are potential compressed EFI drivers that need to be loaded by the EFI core firmware.

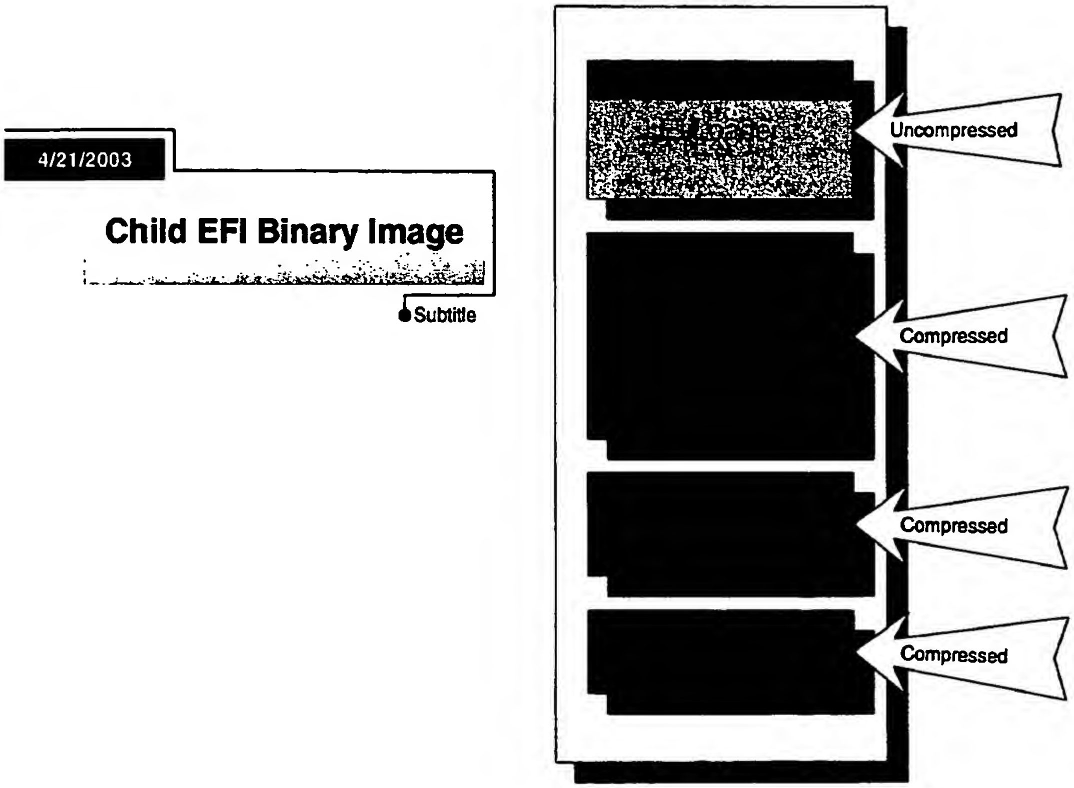


Figure 1 EFI Guest binary Image Structure

This is the memory layout of the last large page in the new EFI child after the switcher is done.

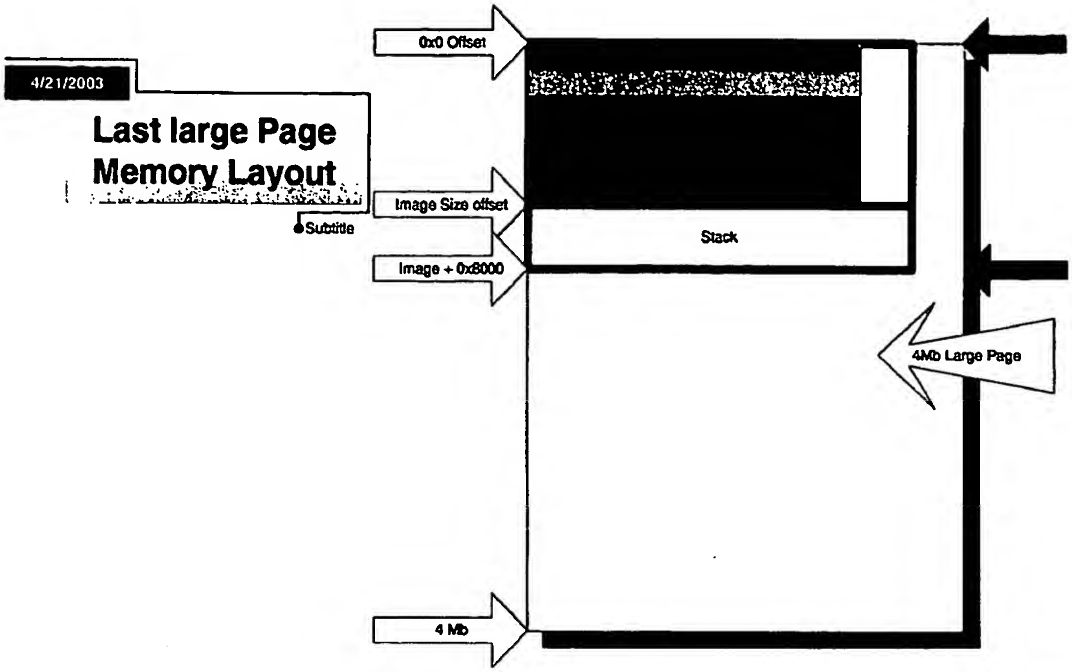


Figure 2 Memory Layout after switcher

First Pass of the Loader

The EfiLoader relocates itself on the first pass and calls itself again.

Diagram:

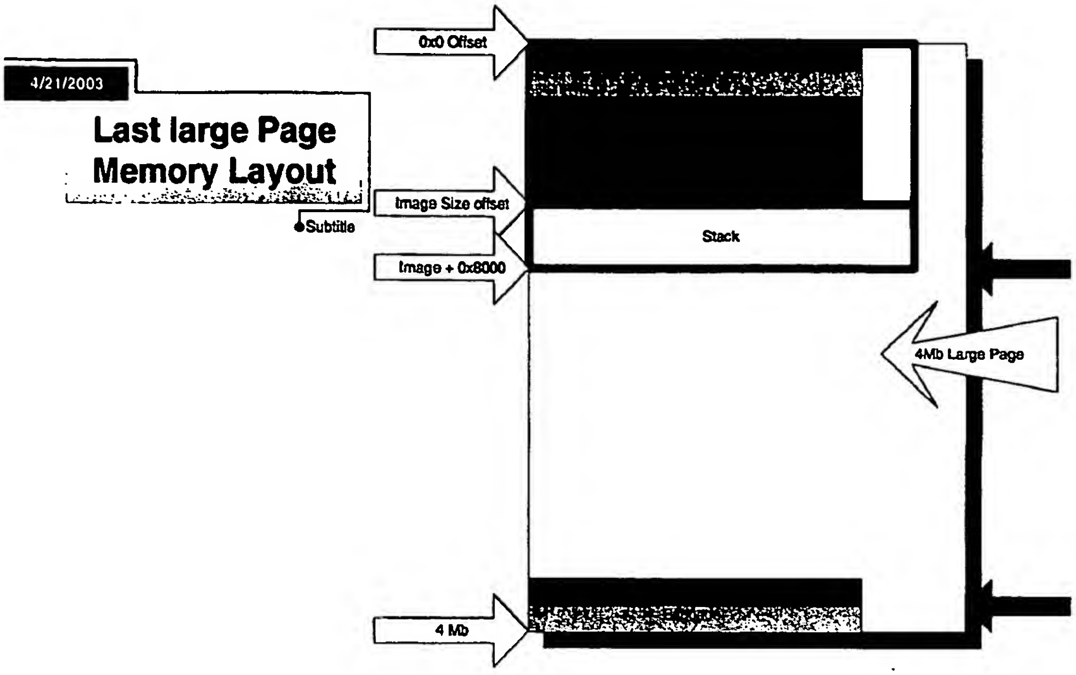


Figure 3 Memory Layout after first pass of the EfiLoader

Second Pass of the Loader

During the second pass the loader creates memory descriptors for the used pieces of memory. Then it allocates a scratch area for decompressing compressed images. Finally it decompresses and relocates EFI core firmware, sets a new stack, and jumps into the initialization routine for the core firmware.

Steps in the second pass of the EfiLoader:

1. Create Memory Descriptor for the relocated EfiLoader image.
2. Create Memory Descriptor for the Scratch area.
3. Create Memory Descriptor for the uncompressed core firmware.
4. Uncompress and place the Core firmware into prepared area.
5. Create Memory Descriptor for the new Stack
6. Change Stack, and jump into core firmware initialization.

Diagram:

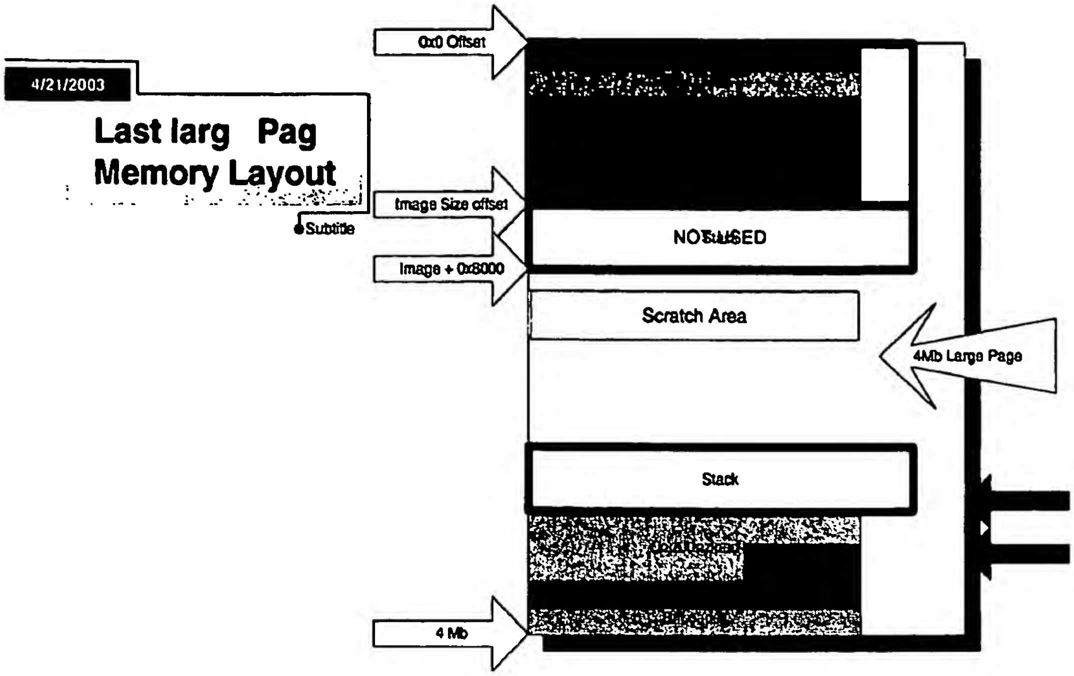


Figure 4 Memory Layout after second pass of EfiLoader

Core Firmware initialization

The core firmware will set up the EFI environment, and load the needed drivers.

Some steps during core firmware initialization:

1. Run VmState.asm to set up IDT/GDT/PD – note that since VmState.asm is part of the core firmware no specific memory descriptor is needed.
2. Create EFI System Table.
3. Load needed drivers.
4. Uncompress and load all of the images concatenated to the EFI guest binary.
5. Remove Scratch area.

Diagram:

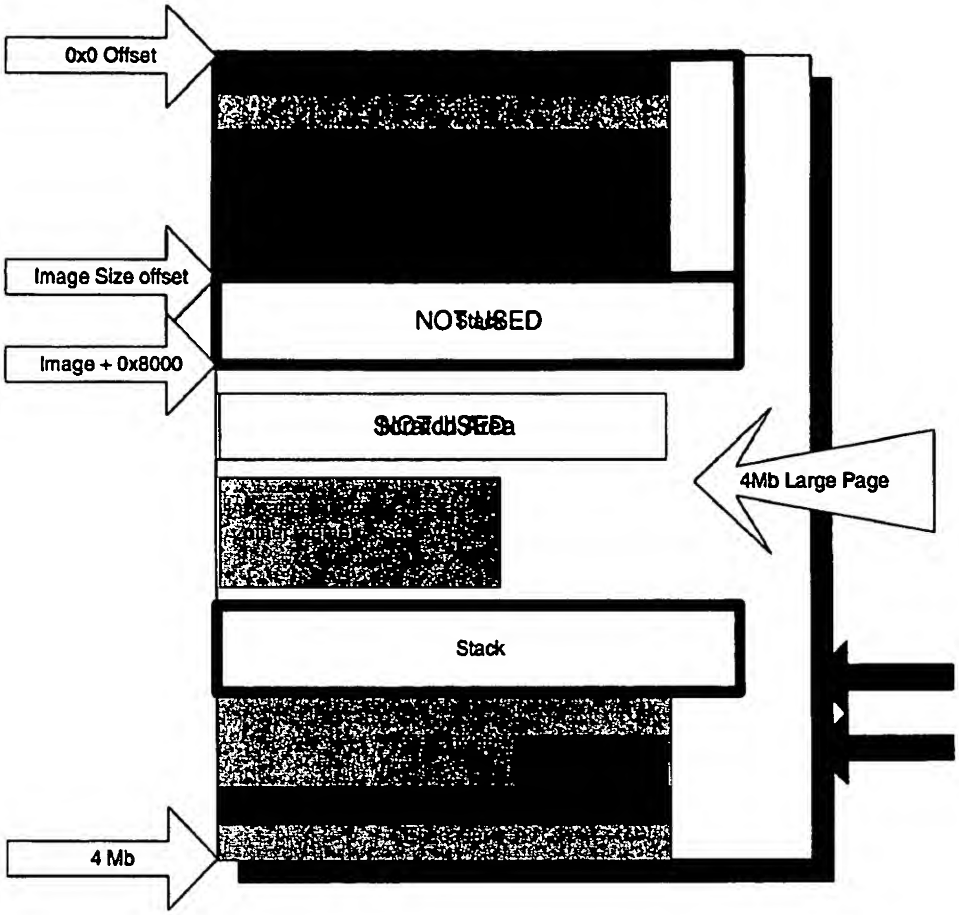


Figure 5 Last page memory layout after Core Firmware initialization

Work Items

This is an approximate estimate of what needs to be done to implement this process

Switcher

- The data structure that abstracts a switched EFI instance will change – it will become simpler.
- The assembly routines to save and restore state will have to account for the GDT, IDT, and paging. (Implemented in stages. In the initial stage the guest EFIs use the PD, IDT, and GDT of the host EFI. The next stages then switch CR3, then also switch IDT, and finally also switch GDT.)

- The interface will change a little.

Guest EFI

- The code that copies and relocates the loader needs to be written (this invokes existing code in efildr.c).(jal)
- The loader needs to be modified to follow the model for the second pass.
- EfiSelfReloc.asm needs to be written (jal)
- VmState.asm needs to be written.

Host EFI

- Efistate.asm (which enables paging) moved from efildr image to the core image as initial entry point. (This is a new 'system' platform driver which includes some C code in a new EfiSystem.C) (This is almost finished – jal)
- Replace IDT from start.asm with one provided by efistate.asm
- Replace GDT from start.asm with one provided by efistate.asm

Future Considerations

- We will need to discuss the mechanism to handle interrupts.
- We will need to finalize the mechanism of communication with CE.

1 Introduction

The Extensible Firmware Interface (EFI) provides architecture for next generation system firmware.

1.1 EFI

EFI goals and requirements are compatible with hypervisor needs. EFI provides structure for hypervisor firmware implementation.

This brief EFI summary is extracted from the EFI 1.10 (.95) specification.

The EFI specification is primarily intended for the next generation of IA-32 and Itanium™-based computers. Thus, the specification is applicable to a full range of hardware platforms from mobile systems to servers. The specification provides a core set of services along with a selection of protocol interfaces. The selection of protocol interfaces can evolve over time to be optimized for various platform market segments. At the same time the specification allows *maximum extensibility and customization* abilities for OEMs to allow differentiation. In this, the purpose of EFI is to define an evolutionary path from the traditional "PC-AT†"-style boot world into a legacy-API free environment.

1.1.1 EFI Goals

Coherent, scalable platform environment.
Abstraction of the OS from the firmware.
Reasonable device abstraction free of legacy interfaces.
Abstraction of Option ROMs from the firmware
Architecturally shareable system partition.

1.1.2 EFI Requirements

Evolutionary, not revolutionary.
Compatibility by design
Simplifies addition of OS-neutral platform value-add.
Built on existing investment.

1.1.3 EFI Attributes

Reuse of existing table-based interfaces (i.e. ACPI)
System partition
Boot services
Runtime services
EFI Driver model

Implemented in C
Defines calling high level calling conventions
EFI driver model, bus/binding protocols
EFI Applications

EFI Drivers (Boot service/Runtime)
EFI OS Loader

Boot only and runtime services provided to drivers and applications
When OSLoader calls ExitBootServices, memory for boot only drivers is reclaimed.

2 Concepts

2.1 EFI Runtime Environment

2.1.1 EFI Images

PE32+

EFI Images are a class of files defined by EFI that contain executable code. The most distinguishing feature of EFI Images is that the first set of bytes in the EFI Image file contains an image header that defines the encoding of the executable image.

EFI uses a subset of the PE32+ image format with a modified header signature. The modification to signature value in the PE32+ image is done to distinguish EFI images from normal PE32 executables. The "+" addition to PE32 provides the 64 bit relocation fix-up extensions to standard PE32 format.

For images with the EFI image signature, the *Subsystem* values in the PE image header are defined below. The major differences between image types are the memory type that the firmware will load the image into, and the action taken when the image's entry point exits or returns. An application image is always unloaded when control is returned from the image's entry point. A driver image is only unloaded if control is passed back with an EFI error code.

```
// PE32+ Subsystem type for EFI images
#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION 10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER 11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER 12
```

Applications
OS Loaders
Drivers

2.1.2

2.1.3 Environment Variables

Boot####
BootOrder
Driver####
DriverOrder
Define one or more drivers that provide runtime services to supervisor.efi application

2.1.4 Boot Manager

Supervisor enhanced boot manager
Assumes normal boot of supervisor, other options provided for diagnostic purposes...

Supervisor.efi defined as boot option

2.2 EFI Development Environment

Windows 2000 or later
EFI 1.1 (1.10.14.60) EDK EFI Driver Kit
Visual Studio 6.0 with Service Pack (SP4 or SP5)
VC++ Processor Pack.
MASM 6.11 (6.13)

24 January 2003 Larry Krablin

There are a number of things a guest operating system or driver must not be allowed do in the Supervisor environment. These are all things that violate the isolation of various virtual partitions in the system, and subvert the Supervisor's overall control. Supervisor must intervene in each of these actions to maintain control and isolation.

The simplest intervention is to force the guest software to run at a privilege level above zero and manipulate things like presence and writeability of memory pages, thus invoking the native protection mechanisms built into IA32. However, the incomplete virtualizability of IA32 still leaves many *isolation violations* uncaught (*virtualization failures*). Even when they are trapped by the architecture, the performance degradation involved in responding to each instance with an interrupt will in many cases be unacceptable.

Other potential interventions involve direct alterations to the guest software, in the source, the object image, or the loaded image. Object image intervention may take place prior to distribution or at install time. Loaded image intervention may take place at load time or at any time before execution of the offending instructions.

A detailed list of isolation violations is provided elsewhere, along with specific interventions that may be used for each one.

One requirement of the interventions is that they must be functionally transparent. At the highest level of abstraction, this means only that users (human interface or application program) of the environment and services provided by the guest software must not be able to detect the existence of either the interventions or the supervisor itself. Moving down layers in the hierarchy, this means that either the guest software itself must be unaware of the interventions or must be in explicit cooperation with them.

At this point, the terms "aware" and "cooperate" get fuzzy. The interventions might be installed by the original producer of the guest software (e.g. Microsoft) or Unisys as the authorized agent of the producer, which would imply very aware and very cooperative. Alternatively, the guest software might be made to be cooperative by forcible changes made to it by supervisor at the object image level, implying awareness and cooperation only at a very low level of abstraction.

Another requirement for interventions is an acceptably low level of performance degradation. Degradation arises both as the overhead of scan and fix, and the cost of fielding and interpreting interrupts.

A series of "food prep" terms are used as a metaphors informally to describe various points in the "intervention space". These range from "raw" to varying degrees of "cooked." The easiest to define are the extreme points of "raw" and "burnt".

Raw:

- no intervention prior to runtime
- boost guest CPL above 0
- scan & fix required for all virtualization failures
- state leakage must be handled (intercepted & faked)
- runtime interventions always triggered by interrupt:
  natural (IA32) protection or INT3
  to identify actual virtualization failures
  to identify not-yet-scanned code
- extensive runtime structure to track scan & fix
- limited lookahead or batching for scan & fix:
  basic block boundaries
  code page boundaries
- guest code addresses, branching structure, etc., unchanged

Burnt:

- no runtime intervention by interrupt at all
- CPL > 0 not required
- hijack of IDT unnecessary
- no object code modification
- all interventions by explicit supervisor calls in guest source

The intermediate degrees of cooked are somewhat harder to pin down, as ther are several dimensions of variability:

- degree of intervention: providing functionality through interrupts or different code vs eliminating the need for the isolation violation altogether
- time of intervention: runtime vs before runtime
- level of abstract of intervention: runtime interrupt, source modification, object modification, component configuration
- performance of intervention

With all that in mind, some stakes in the ground (all versions of cooked less well done than burnt involve raising CPL above 0):

Rare:

- source or object changes to guest to eliminate virtualization failures
- most intervention via interrupt
- state leakage covered by runtime intervention (emulation)

Medium Well:

- state leakage not expected as an issue
- logical calls to supervisor rather than interrupts

Well Done:

- little or no runtime intervention required
- guest makes explicit supervisor calls for services

- state leakage not required

Another item that I'm not sure where to put is handling of legacy stuff that an unmodified guest may fail in the absence of. Component configuration may be the easy way to deal with this. In any case, that may be another flavor added to various cooked options.

1 Introduction

This is the second in a series of informal notes about supervisor, intended for circulation only within the supervisor development team.

"Sensitive" IA32 instructions are those which are not privileged but which yield information about the privilege level or other information about actual hardware usage that differs from what is expected by a guest OS. For example, the pushfd instruction can be used to discover the currently executing privilege level. A typical guest OS expects to be running at ring zero, but supervisor intervenes to actually run the OS at a less privileged ring. If the OS were to use pushfd to directly discover the current privilege level, the OS may malfunction. Supervisor must intervene such that the result of the pushfd is its emulated value (ie, the one established by the guest OS) rather than the current processor setting.

There are 17 or so such sensitive instructions for which a runtime intervention must occur in order to provide an emulated value rather than the actual one. Scan-fix is the process of locating these instructions and inserting the appropriate interventions.

This note discusses "dynamic" scan-fix, a process which occurs as a guest OS is executing. "Monitor" is the subset of supervisor which executes within a guest environment; scan-fix is a monitor function.

Section 2 gives the general idea for scanning and discusses various synchronization and other issues. Section 3 sketches a "ledger" structure, used to keep track of scanned/not scanned and other actual code status. Section 4 outlines key algorithms to accomplish scan-fix.

Some of the items discussed in this note will need integration with other monitor facilities; in particular:

- general page table handling, including pages with "guarded" status
- general Inter-Processor-Interrupt (IPI) mechanism at monitor level, which is likely to include a 'deferred" delivery mechanism of some sort

2 Runtime scan-fix

2.1 Optimization bias

IA32 has both small (4KB) and large (4MB, 2MB for PAE) memory pages. In current Windows, both the kernel and HAL are in large pages, other ring 0 code may be in large or small.

Some general assumptions:

- page faults for frequently executed guest OS ring 0 code will be (very) rare.
- most guest OS code patching will occur during the boot and initialization process.

These imply that most scanning will occur during the boot, initialization and early execution; once up and running actual scanning will be unusual. So performance bias ought to be towards execution of present, previously scanned code.

These assumptions might be wrong:

- if there were "a lot" of page faulting, we might wish to "save" accumulated ledger information; instead of re-scanning when the page is restored we would recall ledger information
- if there were ongoing patching of large pages, we might wish to impose some sort of "subpages" in order to avoid re-scanning of the whole page.

2.2 Two views of code

A guest OS instruction stream needs to be visible to a processor in order to be executed; but we wish to conceal that code from the OS, since it has been patched as part of scan-fix. The scheme for maintaining these two views relies on the IA32 processor having two distinct Translation Lookaside Buffers (TLB) for each virtual address: one which is used for instruction fetches, and another which is used for data access. TLBs essentially hold copies of the information from Page Table Entries (PTE), and include the writable and valid (aka "present") flags. The general intent is that the iTLB be present and the dTLB be not present; the processor can execute through the iTLB but a data access attempt will generate a page fault. Monitor will intervene to do the attempted data access.

In this note, the patched code resides at the virtual address (va) that the OS expects, and a copy of the original code is kept in a monitor-private location (and see 2.15).

2.3 Derived from following execution flow

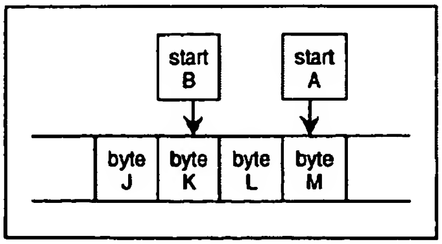
An IA32 instruction stream has no inherent structure other than that which is revealed by its actual execution. For example, there is no architecturally required designation of any byte (or page) as being, or not being, executable code; instructions are variable length, so there's no inherent way of knowing where an instruction begins; sequences of instructions are unbounded, so there's no way of knowing where a particular sequence ends; executable code and ordinary data may be kept in adjacent locations; and so on.

Instead, to find the "sensitive" instructions, scan-fix must work its way through the same sort of decoding process which an actual processor will use. Program "control" transfer instructions[1], called "xfer" in this note, are scanned and handled specially in order to follow this execution flow.

Scan-fix is applied to sequences destined to run at (emulated) ring zero. A transition from a higher ring to ring zero generally occurs via transition through the IDT; monitor intervenes to initiate the scan-fix process on the transition target. During execution within (emulated) ring zero, monitor track xfers as outlined below. A transition from ring zero to a higher ring generally occurs via a "return"; at each return, if the target is also ring zero monitor scans ahead for that target, but for a higher ring it simply returns.

2.4 Multiple interpretations: alignment-misalignment

Any given byte in an instruction stream may actually have multiple possible decodings depending on whether the byte itself is treated as the first byte of some instruction, or whether its is treated as an integral part of some instruction preceding it in the instruction stream. For example, byte M might be interpreted one way if it is viewed as the first byte of some instruction, a different way if viewed as the third



[1] Jumps (aka branches), calls, returns, INT3.

byte of instruction K, and still another way if viewed as the fourth byte of instruction J. While this seems unlikely to occur in normal execution, it needs to be accommodated[2]. The scanner would detect this by noting that either a byte which it views as the first in an instruction has already been seen by the scanner as not-the-first, or vice versa; this is called "misaligned" in this note.

2.5 Intervention

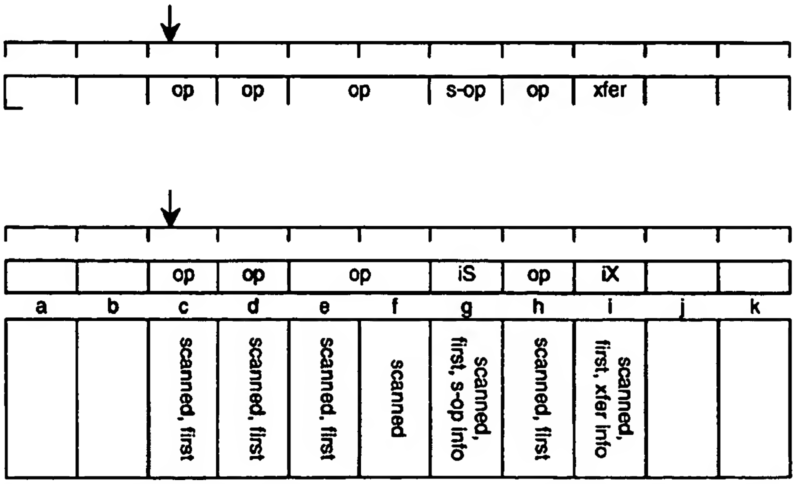
The IA32 INT3 instruction generates a call through the IDT; the instruction is a single byte.

Monitor needs to intervene for each sensitive instruction. This is done be replacing the sensitive instruction in the view of the code actually being executed by a processor. Some sensitive instructions are a single byte; INT3 is used for these. Monitor intercepts the calls through INT3 and so can perform the desired operation.

Other sensitive instructions are multi-byte; where performance is an issue we may look for other interventions, such as placing code inline or using the INTN form to call directly to a particular action. But initially, all interventions will use INT3 to call the monitor INT3 handler.

2.6 Scanning: general idea

Monitor will retain information about its scanning process. This can be seen as the accumulation of a "ledger" entry for each instruction byte, which records (among other things) whether it has already been scanned.



When a sensitive instruction is encountered, it is replaced with the necessary intervention; similarly, when a control transfer instruction is encountered, it is replaced with a xfer intervention. During this process, ledger entries for the scanned bytes are accumulated. For example, scanning for this leg begins with a control transfer to byte c, which is initially unscanned. The scanner is invoked and decodes its way through successive bytes. Byte c is found to be a single byte instruction, and is so marked in the ledger. Byte d is also a single instruction. Byte e is the first byte of a two byte instruction. Byte g is a sensitive instruction; it is replaced in the instruction stream by an appropriate intervention, and the original instruction information is stored in the ledger. Byte h is marked as a single byte instruction. Byte i is a single byte control transfer instruction; it is replaced with a control intervention, and the original information is saved. At this point scanning for the leg is ended. Note that scanning actually consists of a forward phase then a backward phase (see section 2.9.1).

Once a given leg has been scanned/fixed it can be executed at native speed. In this example, the instructions at c, d and e are executed; when g is encountered, control is transferred to the monitor for emulation of the sensitive instruction. Following this, the instruction at h is

[2] The existing ledger is reset (see 2.10) and scanning resumed.

executed. The transfer intervention at byte i returns control to the monitor for evaluation of the transfer.

A scan ends when an xfer is encountered (some sensitive instructions may also terminate the scan), or when a page edge is encountered (see section 2.12).

For each code page which is (partially or fully) scanned, monitor builds a "flag block" of the same size; this is used to hold flags such as "scanned" and "first". In the ledger, this block is associated with the physical page; ie, multiple virtual addresses which map to the same physical page are linked to the same ledger block.

During the forward scan, the flags are monitored as follows:

- if the flags for the beginning of the "next" instruction are scanned-
  - -first, then the scan is ended.
  - -not first then there is a misalignment and the target page is reset
- if not scanned, the length of the next instruction is determined; if the scanned flag is already set for any included byte then there is a misalignment and the target page is reset.

An IA32 "prefix byte" may precede any instruction. It is legal to begin execution of the instruction with the prefix byte or with the instruction itself. During scanning, both the prefix byte and the first instruction byte are marked as "first".

Certain of the sensitive instructions may have prefix bytes. During scanning, both of these are replaced by interventions.

2.7 When the intervention occurs

In the initial implementation, both "sensitive" and "control" interventions will be software interrupts (INT3). This transfer is a "trap" in the IA32 sense that the return frame EIP points just past the byte containing the INT3. The transfer is through a monitor-owned gate in the IDT; because monitor needs to begin with interrupts disabled, this will be an interrupt gate.

By design convention, there is always a ledger flag block for any INT3 placed into the code stream by monitor. So conversely if there is entry into monitor's INT3 handler but monitor discovers that no block is present for that byte, this indicates a guest INT3 and control is transferred to the guest; if this occurs from actual ring zero code, it indicates a monitor bug.

If a ledger is present the flags will be scanned-first (otherwise there's a bug somewhere).

The task is to emulate the original instruction, which is preserved in monitor's private copy of the original code page. The individual cases will be described in a separate note; briefly:

- xfers: this involves insuring that the target is present and scanned, as well as possibly editing the return frame for ring and eflag information.
- others: the return frame EIP is adjusted to point to the correct location in the actual stream, the instruction is emulated; if interrupts were enabled during the emulation then a "scanned return" is used (see 2.10).

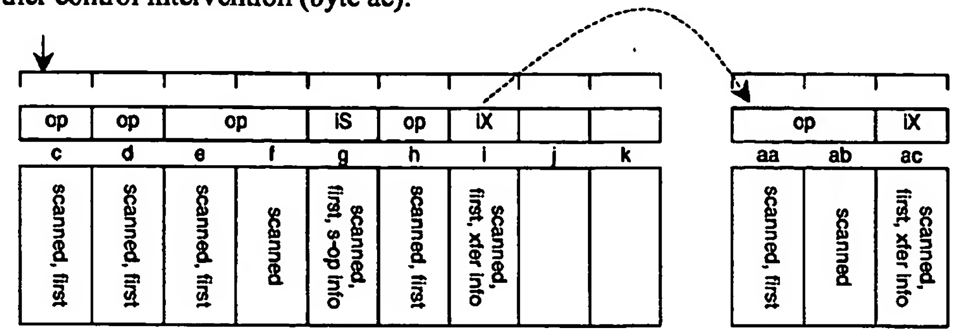
2.8 Optimizing out interventions for branches

The question arises as to whether interventions inserted in order to determine and monitor the execution flow can be later removed. For example:

1 a given leg is scanned-fixed (see example above). The scanner stops scanning the leg at byte i; ie, it replaces the xfer with a control intervention.

2 the leg is executed natively. It ends by re-calling the scanner.

3 the scanner looks at the target of the xfer (byte aa), determines that it is unscanned, and performs the scan, replacing sensitive instructions as needed and terminating the leg with another control intervention (byte ac).



4 the question now arises as to whether the control intervention inserted in place of the xfer in step 1 (at byte i) may now be removed, allowing the xfer from the leg in step 1 to flow natively to the leg in step 3.

This optimization is problematic, since a guest OS may patch execution streams at any time. But any particular write may result in a stream which is not, at the time of the write, actually decodable. For example, it may alter byte aa; at the time of this change, the leg may not form a valid instruction sequence. Any particular leg can only be known by monitor to be decodable at the moment a processor attempts to execute it. Consequently, monitor can't "scan" the stream at the time of any particular write but instead must arrange for it to be re-scanned later. This involves insuring that the scanner is invoked both for the sequential execution case, where the modification is encountered as part of executing the "next" instruction, and for xfers into the modified location(s).

Monitor marks code pages as not writable; if an OS attempts to make a change then a fault occurs and monitor can intervene[3]. A given write can modify multiple bytes; if none of the to-be-modified bytes have been scanned then the modification can be made directly to the code stream; rescan will occur as a matter of course[4].

Control transfer instructions can be divided into two groups: "static" control, in which the target address information comes from (and is relative to) the control instruction itself[5], and "dynamic" control, in which target address information is derived from outside the execution stream[6]. Static xfers have different optimization opportunities: once the xfer has been scanned/evaluated, its target address won't change unless the xfer instruction is itself changed.

[3] Note the problem with a physical IO directly to a already-scanned page replacing the scanned bytes by something else. We may think this unlikely - untypical of a guest OS; however, we will need to decide whether we will design to detect this. This would involve monitoring the target physical addresses of all reads, and intervening if they overlap scanned bytes.

[4] It is possible that executable code and ordinary data are in adjacent locations (ie, in the same page); data writes will be for bytes which are never executed (never scanned).

[5] Jump/call "relative, displacement relative to next instruction"; also jump if condition met rel8, rel16 and rel32.

[6] Jump/call "absolute indirect"; return/ireturn; also far jump, call, INT3.

2.8.1 Static

One way to know the moment at which rescan is possible is to have left in place the intervention call in step 1 (at byte i); on successive executions it's purpose is essentially to check whether a modification has occurred.

For static xfers, we instead exploit IA32 paging. Any given execution sequence is divided somewhat arbitrarily into pages in the IA32 virtual addressing scheme. If monitor were, at the time a page of code is modified, to "reset" (section 2.10) the flags for the page this would insure that ledger information for all in-page static xfers is reset. This would lead to their being re-evaluated the next time they were executed, which would in turn lead to rescanning the modified zone(s). So, for a static xfer, this would allow the control intervention mentioned in step 1 to be removed once it's target leg has been scanned. Interventions for cross-page static branches would be left in place (along with special handling for cross-page transfers which occur within sequential execution from one page to the next (see 2.12)). This approach allows the ledger information for a given page to be discarded; the net effect is that the legs will be re-scanned the next time they are executed.

In this scheme, a control intervention for an static unconditional xfer whose target is within the same code page may be removed once its target leg has been scanned[7].

For conditional xfers, during the initial scan the xfer is replaced by a control intervention. If either leg jumps out of the current code page then the intervention must remain. If both legs are within the current page then at the time the control intervention (at the xfer) is executed the first byte of the untaken leg is replaced with an intervention and the taken leg is scanned; the intervention at the xfer is removed; if the other leg is ever taken, it is scanned and its intervention removed.

Currently, in Windows, both HAL and the kernel are in IA32 "large" pages[8]. If there are ongoing code modifications through the course of normal execution, it may be better (faster) to leave the control interventions in place rather than to invalidate the block for each change. With small pages only the local page would need re-scanning.

2.8.2 Dynamic

Intervention for dynamic execution control transfers ismuch like that for static transfers; the task is to discover whether the target of the transfer has been scanned: if it has been then it may be executed natively, if it hasn't then now's the time to scan it. The same question arises as to what circumstances would allow the intervention placed at the jump to be removed after scanning.

The target address is not taken from the instruction itself during dynamic execution control transfers; instead it is taken from a register or memory location (aka "computed branch"). Monitor can, at that moment, examine its ledger to determine whether the target the has been scanned. But, from an IA32 point of view, there's no particular reason to believe that, the next time that this particular control transfer instruction is executed, it will transfer to the same target. Consequently, it seems like these control intervention can't be removed.

[7] Going further, the scan of the target leg could be combined with the scan which held the unconditional xfer; in the outline above, the scan in step 1 could have simply continued on to scan the leg in step 3 (provided step 3 is in the same code page). Static conditional xfers can't be combined with their target legs in this way, since there's no way to know whether a leg of a conditional branch is intended to be executable until the time of actual execution.

[8] I don't know if this changes with PAE enabled

2.9 Multi-processing

2.9.1 Read/execute while scanning

Since significant guest code is likely to be in large pages, it seems desirable to allow execution of already-scanned legs in a page even though some other area of the page is being scanned[9]. This means that the scan process must maintain a consistent view for any readers; ie, at every moment all possible xfer targets are either:

- not scanned (ie, appropriate flags in ledgerblock, or no ledgerblock for that target), or...
- are sequences whose first byte is scanned; in which there are no sensitive instructions (there may be sensitive interventions); and which is terminated by an xfer intervention.

Various steps are taken to maintain this in a multi-processor environment.

Firstly, scanning must be occurring on only one processor at a time; associated with the ledger will be an appropriate spinlock, which must be held by the scanner.

Secondly, during scanning the ledger must remain consistent as observed by other processors.

To maintain consistency, the scanner will scan forward to the "end" of the leg, then build the ledger and interventions backwards to the initial target. The scanned flags are left alone during the forward phase, then set during the backward phase after the page alterations for that particular byte have been made. Other processors which consider this zone as a target will observe either the correctly formed "tail" of the leg[10], in which case it may be executed at native speed, or will observe a unscanned target, in which case they will enter the scanner (and stall).

In some circumstances it isn't possible to maintain a consistent view; it these cases an Inter-Processor-Interrupt (IPI) is used to drive any readers out of the target page (see Reset, 2.10).

2.9.2 Scanning by page

During boot there are likely to be various ranges of executables being established, particularly as various guest drivers are loaded. So it seems desirable to allow a scanner per page rather than only a single scanner; an individual scanlock is associated with each flag block.

2.9.3 Changing root

Changes to the root ledger structure are guarded by a "root" spinlock. But acquisition of this lock by itself does not insure that there are no readers or scanners active.

Additions to the structure utilize atomic store ordering such that, after appropriate initialization, a single store is used to make the new data visible.

Deletions require that there be no active readers or scanners, and so involve an IPI-response transaction. The general meaning of this IPI is "stop using the ledger, then re-evaluate its status". The deleter will send this IPI to all processors which may be active in the ledger (ie, all processors assigned to the VM); it then waits for a response from each of them.

If the IPI arrives at a processor which is not at that moment executing monitor (ie, not using the ledger), the interrupt handler can generate the expected response, then exit back to the interrupted sequence.

Monitor's IDT slots are initialized such that during entry to monitor external interrupts are disabled. There are only a few key moments when monitor's scanning routines are prepared to

[9] I'm mostly concerned with boot here: an implementation which required that no readers be executing while a scan is occurring would require an IPI-response-response sequence for each leg.

[10] Not to mix a metaphor...

act upon a "stop" request. At these moments interrupts are explicitly enabled/disabled; this is always followed by re-evaluating the ledger status. Typically, a monitor sequence would not enable interrupts at all, and the interrupt delivery would be deferred until it returns to the guest.

In order to initiate the IPI sequence, a processor must be holding the root lock; so during a loop which is attempting to obtain the root lock, interrupts must be enabled/disabled. Further, an initiator of an IPI sequence must not be holding any lock which may be needed on another processor (eg, no scanlocks).

Of course a momentary interrupt by itself won't prevent new usage of a ledger structure. To do a deletion involves a sequence something like:

- get the root lock
- "null" the pointer to the structure (blocks new entrants)
- generate the IPI
- wait for all responses

2.9.4 Verify these assumptions?

Allowing readers while scanning, and allowing multiple per-page scanners both add complexity to the Monitor. I've included them in this design sketch. We might wish to measure the actual performance benefits, and if it is not significant then simplify the design.

2.10 Reset

Using the underlying memory pages as a framework allows optimizing out interventions for local static xfers once their target legs have been scanned (section 2.8). When a guest attempts to write into a code page, the associated flag block is "reset"; ie, the actual page content is restored, and the associated flags reset. During this process there mustn't be readers or a scanner using the page; an IPI is needed to arrange this.

The page is restored to a state which reflects the original page content plus any previous writes done by the guest; this data is kept current in the ledger structure (and used for guest reads).

The reset function is also used when a misalignment is detected (section 2.4), and when the guest "discards" the last va-pa mapping for a given physical page.

2.10.1 Execution barrier for previous page

As noted above (2.8), xfers into this page from other pages involve a control intervention, and this will take appropriate action for a page which has been reset. Sequential execution from the previous page into the page being reset requires its own handling.

Generally, scanning is serialized, and the same serialization is used during reset. This implies that, when the serializing lock(s) is obtained the current ledger status is consistent. So if the ledger shows that the first byte of a being-reset page has not been scanned then there's no special handling needed; the page may be reset. Similarly, if the first byte of the reset-page has been scanned, but a) the previous page[11] is OS-absent, or b) the last byte of the previous page has not been scanned, then no special handling is needed.

If the previous page is OS-present and the last byte has been scanned then monitor locates the last scanned/first byte in the previous page and changes this to a "edge" intervention (see 2.12); after this, the page may be reset

[11] "previous" refers to the page mapped to the va previous to the va of the page being reset.

2.11 Scan-return

During the time when we are executing monitor code for an intervention or interrupt, we are in effect holding a non-local entry point into the guest code. If the monitor routine is such that it enables interrupts (see 2.9.3) then during the routine the monitor status of the guest page could change: it's flag block could be reset, and the guest could have made the page absent. So as a part of returning from such a routine, monitor must check the status and take appropriate action; briefly:

- if the actual page is guest-absent, induce a page fault (see 2.13)
- if the page is present but there's no ledger, allocate the ledger
- if there's a ledger and the return point...
  - o is scanned-first, then just return
  - o is scanned-not first (ie, misaligned), then reset the ledger, scan, and return
  - o isn't scanned, then scan and return

If it's necessary to induce a guest page fault, it's something of a riddle just what return-eip to use. In IA32, the return-eip should point to the instruction whose execution detected the absent page[12]. But here we are detecting it explicitly as a monitor artifact, and there's no particular guest OS instruction to implicate as the absence-detector. So monitor will use the address of the absent instruction as the return-eip.

2.12 Cross-edge execution

Handling scanning for a cross-page edge is complex since two separate pages are involved. The multi-processing handling for this is somewhat different than for a single page, so treating cross-page differently then single page allows the more usual case to be simpler.

When the forward phase of scan comes to a page edge, it will terminate the current leg. The last byte of the leg will the last byte which is the first byte of an instruction (called the last-first). This byte is replaced with an "edge" control intervention, then the remainder of the leg completed. The leg is then executed at native speed, ending with re-entry to the "edge" intervention handler.

At this point the next page may be OS-absent, or it may be OS-present but not have a ledgerblock, or it may not yet have been scanned, or it may have been scanned but is misaligned with the previous page. Concurrently, Monitor executing on another processor may be trying to de-allocate either page, or reset either flag block, and so on.

The root lock is used to coordinate this; root is acquired at the beginning of the edger process.

Generally:

- if the next page is OS-absent, a page fault is induced.
- if the va has no flag block, a block is obtained
- if the first byte(s) of the next page is misaligned, its flag block is reset.
- if the first byte(s) of the next page is not scanned, it is scanned
- the "edge" intervention is removed

[12] In typical guest os execution, if a processor is executing from a page which is made absent by code running elsewhere, the processor just keeps executing until some action occurs which invalidates its iTLB (such as jumping to another page whose iTLB entry displaces this one); the guest insures this by sending processors an IPI before actually re-using the forgotten page; if this interrupts the sequence in the forgotten page, the absence is detected on the interrupt return, and the return-eip points there.

2.13 Page fault f r OS-absent page

In IA32, a page fault interrupt has two interesting addresses: the address of the item which is absent (CR2), and the address of the instruction whose execution caused the absence to be noted (return-eip). For a page fault which occurs due to absent code, CR2 points to the missing page and return-eip to the instruction which couldn't be executed. The intent is that software act to make the page present, then return to that eip; the instruction will be re-executed.

For an xfer intervention whose target is in an OS-absent page, the return-eip is set to point to the xfer intervention.

For an edge in which the second page is OS-absent, there are two cases:

- the last instruction in the first page overlaps the edge; eg, a 2-byte instruction whose first byte is the last byte of the first page. In this case, the return-eip points to the last instruction in the first page.
- the last instruction in the first page ends exactly at the end of the page. In this case, return-eip should point to the first byte of the second page.

This second case adds complexity to monitor: the puzzle is how to execute the last instruction without at the same time (due to asynchronous action which makes the page present) allowing the first instruction of the next page to be executed unscanned.

In this note special action is taken when any page is made present (by make present I mean that the guest OS is attempting to load a PTE with the valid bit set). Monitor intercepts this for several reasons (eg, to convert the b address to a c address). This routine will get the new va and check to see if the va for the previous page has a flag block for a physical block in which the last scanned instruction ends at the edge; if so, it resumes scanning at the first byte of the new page. With this scheme, a last non-overlapped instruction in a page can be executed at speed.

This is somewhat messy to implement, and I'd welcome other proposals. I've thought some about:

- *emulating the instruction; since this can be any valid IA32 instruction, this would require that there be a full-blown emulator as a part of monitor, which is something we don't seem to otherwise need.*
- *executing the instruction "natively" from some other more "controlled" location: but if the page fault does occur, the return-eip will be wrong; also, if the instruction is a call, or relative branch...*
- *using the hardware to single-step the instruction: this would not prevent other processors from executing the instruction at speed.*
- *don't execute the instruction; induce the page fault with return-eip pointing at the last instruction in the first page. As I understand it, an actual processor wouldn't generate this case, but the question is: does the OS page fault handler care? I don't know how to answer this...*

I'd welcome other views...

2.14 Guarded pages

Monitor must intervene at any attempt to read a page which has modified code in it; such a page is "guarded" in the sense that the actual PTE (used by the processor) is marked absent even though the guest status of the page is "present".

A particular page comes to monitor's attention as "code" the first time there is an attempt to execute from it, typically detected during a control intervention; this leads to the allocation of a ledger for the page. From this point forward (until deallocation), monitor needs to intervene for each attempted read or write. These references could be made via the same va-to-pa mapping through which the execution attempt was made, or through a different va-to-pa mapping constructed either before or after the initial execution attempt.

This means that at the time of the first execution attempt monitor must search for existing va-to-pa mappings which point to the physical page being touched, and make them "guarded"; it also means that every mapping attempt occurring later must be vetted against existing guarded physical pages. Further, when the last code mapping for a given physical page is discarded by the guest, any other mappings to the same page need to be returned to "unguarded" status.

The details of these actions are TBS; some notes:

- at the first execution attempt, the method for finding existing mappings is likely to be dependent on the overall handling of "shadowed" page table information.
- of course it will be (much) faster to search only the tables for entries the guest has marked "supervisor". We've already (more or less) taken the position that we aren't virtualizing for a guest's "user mode"; for example, if user mode code does an SGDT we are not planning for monitor to intervene, instead the user will get a monitor value and not the guest emulated value. So perhaps we can treat an attempt to read a ring zero page similarly - leave the user mapping unguarded and let the user read the (possibly patched) code (I don't know whether this would bother the debugger). For writes, I have to wonder under what circumstances there would be a user-mode writable PTE for ring zero code (debugger?).
- if we discover existing mappings, we are likely to need IPIs to invalidate TLBs on other processors.
- a mapping becomes guarded because the physical page is guarded (because it has a ledger and possibly patches). When the guest attempts to set a page table entry to "present", it's the offered physical address that is checked against some sort of list of guarded physical pages. There are other reasons why monitor may similarly "guard" physical pages; for example, the pages representing memory-mapped IO space, or the page tables themselves; this suggest some sort of consolidated list of guarded pages.
- when a page is to become unguarded, we could again search; or we could accumulate a list of references.

2.15 TLB shuffle

The machine will actually be executing code which has been modified as needed for scan and fix. So we are interested in preventing the guest from reading the actual code area; when a read is attempted we wish to intervene and present the original code. Similarly we wish to intervene on an attempt to write into an actual code page so we can update scan-fix status for the altered region.

In current Intel IA32s, Translation Lookaside Buffers (TLBs) hold copies of entries from the page tables; current CPUs have different TLBs for code than for data. Monitor manipulates references such that the iTLB is different than the dTLB: for a given PTE pointing to ring zero code, generally the iTLB is present but the dTLB is absent[13]. This means there will be a page fault each time a guest attempts to read or write the code, and monitor can intervene at this point.

During its usual scan, monitor can detect the first touch of any particular code page, can mark its status, and can take the steps necessary to establish the two TLB views.

[13] The IA32 architecture does not "require" the presence of TLBs, nor does it "require" separate code and data TLBs. However, we have not discovered another feasible method for handling code such that it is executable but not readable, which is how we prevent the guest from discovering that we've modified the code.

The content of a TLB slot can discarded at any moment by the actual processor. On the next attempt to execute the actual code associated with the slot the actual PTE will be loaded, then a page fault will occur; monitor can then take steps to reload the TLBs.

In order to establish the two views, the steps are:

- patch a "return" into the target page
- flush the TLB (discards the "absent" PTE from TLBs).
- mark the actual PTE "present"
- call the return (loads the "present" PTE to iTLB but not dTLB).
- mark the actual PTE "absent"
- put the original instruction back

It is problematic to patch a return[14] into the target page, since a given target page can be in use on multiple processors. One approach would be to use an IPI transaction to "stall" other processors during this processor's modification. This approach would have a high performance cost: there's the IPI handling cost, and also all processors are stalled until the last one checks in. Instead, monitor makes the modification on-the-fly.

A "return" instruction is a single byte. Monitor (with appropriate serialization) can safely patch it into any unscanned byte, execute it, then repair the target page. If all bytes in a page are scanned, monitor locates any scanned instruction which is two or more bytes long. The first byte is patched with a "bump" intervention; any processor executing sequentially into this byte will enter the monitor and re-evaluate. The second byte is patched with the return, then executed. If the target bytes are in the same (atomic) store target then they can be patched simultaneously; otherwise the first byte is patched then the second, the return is executed, the second byte is repaired then the first.

Note the small window during which the PTE for the modified code is marked "present"; if another processor/guest were to do a data read at this exact moment it could see modified code.

3 Tables

Once scanning is more or less complete, the ledger is used mostly during sensitive interventions, and during control interventions for dynamic branching. As a performance goal, it should be possible to read the ledger without locking; writing the ledger will involve locking, and must be devised to allow lock-free reading.

We need a ledger "flags" entry for each possible executable byte. The IA32 architecture allows for any address to be an executable byte, so it isn't feasible to use a direct mapping of the execution va into a ledger structure. Instead we'll use some sort of "lookup" structure at the ledger root, in which an entry points to a ledger block[15]; each entry at the root corresponds to a va range in which there is code.

[14] A guest "return" is itself an xfer and so is replaced during scanning with a control intervention; so we are unlikely to be able to find a guest "return" somewhere which we can just "borrow".

[15] Lookup in this structure will be a significant performance concern; I've described here a two level structure, but we may find a more optimal organization after actual prototyping.

A flag entry (flagblock) is built at the time a control intervention detects that there is no ledger for the target of an xfer. At this moment only the size of the page holding the target is known; the size of the range in which that page occurs is not known. For example, a given binary might occupy 50 contiguous 4K pages, but the initial xfer is to only one of those pages. It doesn't seem desirable to make a root entry for each of those pages (as they encountered during execution), since this would add 50 entries at root; this would slow down lookups through root. Conversely, the ledger needs to be mapped into guest OS va space[16], so allocating ledger blocks which are "too large" would unnecessarily consume guest OS va space. We will need to do some experimenting here; as an initial scheme we'll allocate a full "large" block for "large" pages, and perhaps a 64KB[17] block for the first entry to a 4K page, at the appropriate modulus.

A flag block is associated with a physical page. A physical page is the unit of allocation/deallocation a guest OS will use; an individual flag block will be the size of the associated OS page[18]. IA32 allows multiple vas to be mapped to any physical page, so a given flag block may have multiple vas which use it. When a new va is to be used for execution, monitor needs to associate that va with a flag block. Monitor uses the pa associated with that va to see if there is an existing flag block for that pa; if so, it is linked otherwise a new flag block is created.

We are maintaining an environment in which both reads and writes to code pages by a guest OS generate faults, which allows monitor to emulate/propagate the desired access. But monitor itself needs read/write access in order to do scan-fix. So a code page will be mapped into the guest va space[19] a second time by monitor.

We need to save the original code so that it may be used during a sensitive intervention, or a guest attempt to read or write; guest writes are propagated to this copy.

Summary

A va becomes interesting when it is the target of an execution attempt. On the first execution attempt the root and subranges are added to the ledger, the PTE is given shuffle status; if the associated physical page is not already guarded it is made guarded and the associated flagblock and other structures are constructed.

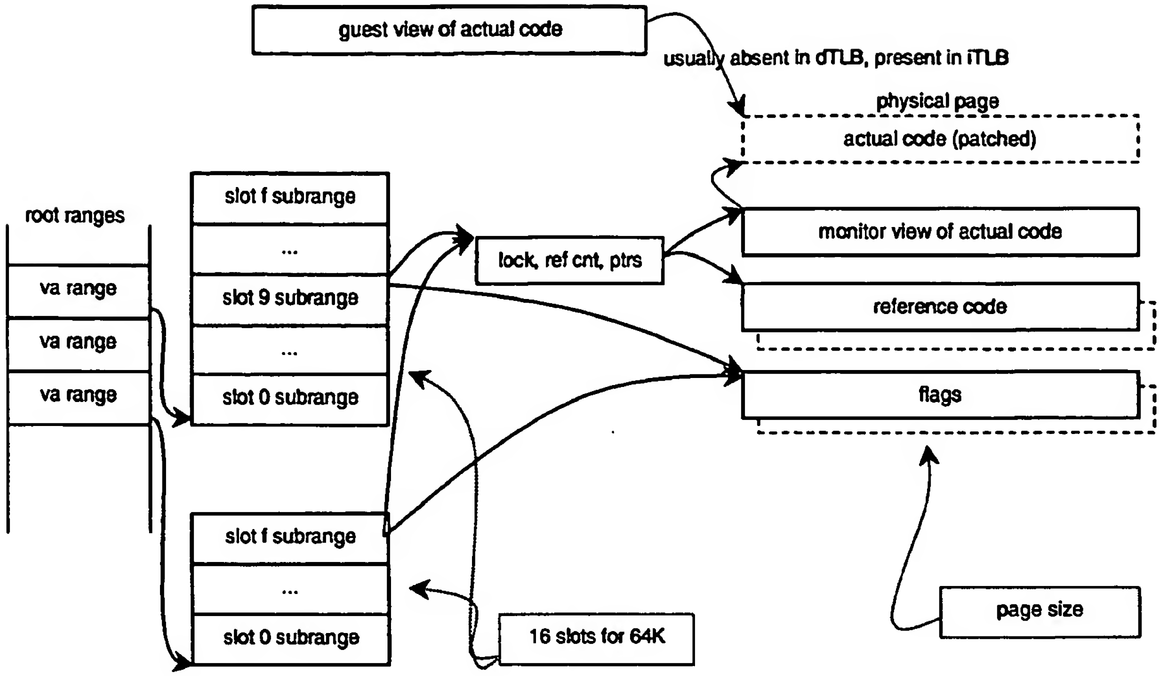
A va becomes uninteresting when a guest (attempts to) resets the associated PTE's "valid" bit. The subrange entry will be withdrawn; if it's the last subrange for a root range then the root range will be withdrawn. If it is the last reference to its associated flagblock then the actual page will be restored and the flagblock (and related stuff) withdrawn.

[16] if we "float" this (ie, dynamically remap virtual addresses onto different monitor structures) we'll need "too many" TLB invalidations.

[17] as I recall, Windows does some consolidation here; for a page fault both the requested page and some "surrounding" pages are made present; I don't know that we can exploit this...

[18] In initial implementation, this is also the size used for optimizing out page-local static branching. In the unlikely event that during normal execution the guest OS makes ongoing patches to "long" pages, we may wish to implement a scheme in which a long page is subdivided into multiple branch-optimization frames.

[19] borrowed or claimed



The structure is similar for a large code page, except the subrange block has a single entry.

4 Routines

The following sections outline details of the steps needed to monitor the execution flow and insert interventions for sensitive instructions. Some of these routines are likely to share information with other monitor routines for general handling of IPIs, and for page tables; these details are TBS.

Some synchronization notes:

- *scanlock allows altering a visible ledger; readers are active (maintain consistent view)*
- *holding root allows adding/altering root table and subrange contents; there are readers/scanners active (must maintain read-consistent view)*
- *a range/subrange/page/ledger won't go away if a routine is ignoring interrupts; to take away, need root, scanlock (for spinners), hide (readers) then ipi-spin to wait for them to leave*

4.1 int3 handler

every monitor INT3 always has an associated ledge; an INT3 is a

- *control intervention: go to the control intervention routine*
- *a page edge intervention: go to the edger*
- *a sensitive instruction: go to the sensitive instruction emulator/handler (not discussed in this note)*
- *a bump; ie, we ran into some code at the moment the shuffle routine was patching it to load iTLB*
- *an INT3 that isn't Monitor's: go to the guest OS INT3 routine*

```
get calling addr from frame eip
if there's no ledger for the calling addr, it isn't ours; go to guest OS
there is a ledger; it won't go away while we're ignoring interrupts
if target is scanned/first
    if edger flag  go to edger
    if sensitive flag  go to emulator
    if not bump flag  go to control_intervention
scanner updates the code, then the flags, leaving timing window here
lock scanlock
assert(target is scanned/first)
temp=flags   these won't reset while we're ignoring interrupts
unlock scanlock
if temp.edger flag  go to edger
if temp.sensitive flag  go to emulator
if temp.bump flag  return
go to control_intervention
```

4.2 control_intervention *follow execution flow to scan ahead*

at entry, every interruption has an associated ledger, which is used to get the right action.

some conditions we encounter require more locking than others, this routine is somewhat repetitive in order to mainstream the "usual" case, which is the first few lines

```
decode target address
temp = get_ledger_adr(target)
if it has a ledger   implies OS_present
    if target is scanned/first
        replace frame eip with target address
        ireturn
    lock scanlock
    if target is scanned/first
        replace frame eip with target address
        unlock scanlock
        ireturn
    if target is not scanned
        temp=scanner   returns 'valid', or 'reset' if it hits a misalignment
        if temp == valid
            replace frame eip with target address
            unlock scanlock
            ireturn
        scanner hit misalignment
    initial byte is scanned/not first; ie, misalignment
    unlock scanlock
no ledger, could be hidden, or OS_absent page
get_root
note that while getting root, the calling page may be discarded; we can still use the original frame if we need to generate a page fault on the target;  it's also possible that the ledger has come or gone, been scanned or reset, etc.  so we generally start over, but holding root.
if target is OS_absent   fabricate page fault
    backup eip, fix frame, set cr3
    unlock root
    far jump to os page fault
if no ledger temp=allocate_range(va,pa); subrange.pflags=temp;
else lock scanlock
temp=scanner;
if temp != valid misalignment
    reset_ledger(this)
    temp=scanner(this)
    if temp != valid halt
replace frame eip with target address
unlock scanlock
unlock root
ireturn
```

4.3 scanner *parses actual code sequence*

caller has scanlock

scanner first scans forward to find the end of a leg; a leg ends at an xfer, or at the end of a page. during forward scan, alignment is checked; if a misalignment is encountered, the scanner quits and returns a "reset" flag. Otherwise, scanner moves back through the leg, patching into the actual stream and updating the flagblock (in that order); scanner -must- maintain an externally consistent view, using properly ordered atomic stores.

```
note that scanner may be called with initial target byte
    scanned/first: return 'valid'
    scanned/not first: return 'not valid'
    not scanned: do the scan
as a part of setting scanned/first/sensitive/edger flags, reset bump flag
```

4.4 reset_ledger(pa) *discard "scanned" status when, for example, guest patches code*

```
caller has rootlock and scanlock for target ledger/page
if first byte of page is scanned
    we need to place barriers at each page which execute sequentially into this one
    for each subrange that points to this pa
        if there is a ledger for the previous (lower va) subrange
            lock(previous scanlock)
            we're holding root, so a ledger can't disappear
            if there's a last-first  start with the last byte, step backwards: if not s then quit; if s/f this is it
                if it isn't an edger intervention
                    put an edger there
                    set the special ledger flag for this byte must be single store (readers active)
            unlock(previous scanlock)
okay, we've blocked sequential; now get the readers out of the ledger/page; we can't be holding any scanlock
during IPI (some responder may be spinning), so we temporarily hide this ledger block; when we do the ipi this
will drive out readers (they'll be spinning on root)
turnoff(this)  finds each subrange that points here; must be single store
unlock this scanlock
IPI et al
spin waiting for all responders
lock(this scanlock)
now there's no one accessing the ledger/page (there could be scanlock spinners)
copy original code back to actual page
clear all ledger flags
turnon(this)  finds each subrange that points here; must be single store
return
```

4.5 edger *handle control flow and scanning across page edge*

at an edge, usual scan will stop scanning and put an edger as an intervention into the last(first) op; when this intervention is executed we come here to check page presence, alignment, etc

when we enter here, the calling page/ledger is present and the last-first is s/f; the "next" page may or may not be present; generally, disabling interrupts keeps a page from going away; here we need to hold root to prevent one from appearing (and for reset); but getting root allows the calling page to disappear. So after we get root we need to re-examine the calling page status

"this" refers to the calling page, "next" to the next sequential page

```
get_root   includes enable-disable loop; doesn't block readers or scanners
if no ledger for this page then unlock root, scan_ireturn   someone has reset the calling ledger
if my_eip is not s/f then unlock root, scan_ireturn  someone has reset the calling ledger
if instruction at my_eip isn't edger then unlock root, ireturn page was replaced/rescanned
```

```
we get here (eg) if, during the root lock sequence, nobody messed with the calling page
overlapped is true if the last instruction in this page is not complete (ie, does not end on the page end)
lock this scanlock
if next page is OS_absent   this is a check on the next sequential va's pte
    if overlapped
        last instruction overlaps the edge; page fault should be for that instruction, so we induce it here
        convert current frame to page fault
        unlock this scanlock
        unlock root
        far jump to guest page handler
    it's not overlapped; ie, the page fault should occur at the edge; here we'll smash into the edge and hardware
    will generate the page fault.  later, when the page is made present, we'll intervene to scan from the first byte
    if the original byte was sensitive, put an intervention in the actual page
    otherwise put the original byte back
    reset the special ledger flag for this byte must be single store, last (readers/scanners active)
    unlock this scanlock
    unlock root
    if there were a delay here, some other thread could make the page present; prior to scanning, a thread could
    execute past the end of this page into unscanned material; makepresent handles this by, if necessary,
    resuming scanning at the first byte before making the page available
    iretum
the actual code still has our edger intervention, which keeps new readers from executing into the page before
we've checked its alignment (and maybe reset the ledger) and scanned if needed
if next va doesn't have a ledger temp=allocate_range(va,pa); subrange.pflags=temp;
else lock next scanlock
temp=scanner(eip)  scan into the next page
if temp != valid   misalignment
    existing ledger analysis is mis-aligigned with this entry point; reset the ledger block; we can't be holding any
    scanlock during reset (some responder may be spinning), so we temporarily hide this ledger block; when
    reset does the ipi this will drive out readers/scanners (they'll be spinning on root).  reset_ledger will
    temporarily hide next ledger
    tumoff(this)  finds each subrange that points here; must be single store
    unlock this scanlock
    reset_ledger(next)
    lock this scanlock
    tumon(this)  finds each subrange that points here; must be single store
    temp=scanner(eip)  scan into the next page
    if temp != valid  halt
replace edger intervention with correct byte  must be after scan
reset s/f and special ledger flag for this byte must be single store (readers active)
unlock next scanlock
unlock this scanlock
unlock root
iretum
   Note: In handling the scanning across a page boundary we need the scanlock for two pages, at successive
   vas.  A scanlock is actually associated with an underlying physical page.  If both of these vas were mapped to
   the same physical page we'd try to get the same lock twice. Further, the edger code relies on being able to
   control the ledger flags for the first page independently of the second, consequently it won't work if the second
   page is reset due to a misalignment.  Coding around this is problematic, and the double sequential mapping is
   ridiculous, so for now, detect and halt.
```

4.6 makepresent(va,pa) *guest OS sets PTE "valid" (makes present)*

fault on a write to OS page table, guest is adding a mapping; see if we have a pending unscanned non-overlapped edge. for an overlapped instruction, the scan resumes from edger; here, we're looking for the case where the last instruction of the previous page has already been executed, and ended right at the edge.

```
    get_root
    acquire guarded status from physical page guard status  -  TBS
    if previous va doesn't have a ledger
        unlock root
        go notreally
    previous page is present, mapped
    if last instruction of previous va page isn't scanned or doesn't end at page edge
        unlock root
        go notreally
    temp=allocate_range(va,pa)  might link to existing page/ledger
    temp2=scanner
    if temp2 != valid   scanner hit misalignment
        reset_ledger(pa)
        temp2 = scanner
        if temp2 != valid   scanner hit misalignment
            if temp2 != valid   halt
    once it's updated reader can use it; must follow scan and precede subrange update
    note that 'valid' is set in the guest 'PTE', but not set in the actual PTE (this triggers shuffle)
    subrange.pflags=temp
    unlock scanlock
    unlock root
    go around
notreally:
    ...
    scan_ireturn
```

4.7 makeabsent *guest OS resets PTE "valid" (makes absent)*

fault on a write to OS page table, guest is deleting a mapping;. we need to block sequential execution from the previous va page. if this is the last mapping for a given page, discard the subrange.

```
    get_root
    subrange=get_ledger_adr(target)
    if subrange is null
        update pte
        unlock root
        scan_ireturn
    lock this scanlock
    turn off just this subrange   block new scanners/readers through this va
    if first byte of this page is scanned
        lock previous scanlock
        if there's a last-first  start with the last byte, step backwards: if not s then quit; if s/f this is it
            if it isn't an edger intervention
                put an edger there
                set the special ledger flag for this byte  must be single store (readers active)
        unlock(previous scanlock)
    get the readers out of this ledger/page
    unlock this scanlock   may be someone spinning
    IPI (this) et al
    spin waiting for all responders
    clear out subrange
    if all subranges in this block are now empty
        deallocate root range entry
    if last mapping for this pa
```

```
        unguard other PTEs - TBS
        deallocate auxilliaryblock
        copy original code back to actual page; deallocate copyblock;
        deallocate ourview
        deallocate flagblock
    update pte
    unlock root
    ...
    scan_ireturn
```

4.8 allocate_range(va, pa) *returns flagblock pointer*

```
    caller has rootlock
    returns with scanlock held, link hidden; this let's caller diddle before making the flagblock visible
    if va isn't in root
        allocate associated subrange block   flagblock pointer must be null
        build root range entry   during construction, must always be sensible for readers
    hunt for existing ledger block for this pa
    if found
        a new mapping; some other mapping already exists
        lock scanlock
        subrange.pflags=null
        subrange.paux=existing aux block
        return (flagblock addr)
    first touch of code at this pa
    allocate auxilliaryblock;
    aux_shuffler=null
    lock scanlock
    guard this PTE - TBS
    hunt down existing PTEs for target's va: guard them - TBS
    allocate copyblock; copy actual code to copyblock
    allocate ourview (present/writable) va block for this pa
    allocate flagblock   must be zero-filled
    subrange.pflags=null
    subrange.paux=existing aux block
    return(flagblock adr)
```

4.9 get_ledger_adr(va) *returns null or address of flags associated with actual code va*

```
    note that the flagblock pointer may be nulled at any moment, but the other stuff stays valid while we're disabled;
    ie, if we see it non-null then we cam use it, but don't look twice.
    look through root table for entry which encompasses va; if not found return null
    use masking info from root to compute location of subrange for this va
    temp = address of flagblock
    if temp is null   return null
    use masking info from root and temp to compute location of flag byte in flagblock for this va
    return flag addr
```

4.10 scan_return *special monitor exit*

enabling interrupts in monitor scan-fix code (needed to get root lock) allows ledger resets to be processed from other processors; this may affect the page from which this monitor intervention occurred. similarly, while executing guest code we may get an interrupt for a ledger reset. in these cases, before we return, we need to check the code we will return to see if it now needs scanning. this routine is much like control-intervention, except the target address is taken from frame eip. An unsolved riddle is what eip to use if we need to induce an OS page fault.

4.11 get_ro t *locks indexing structures*

the root lock governs (among others) IPIs. to prevent deadlock, the acquisition loop must itself process incoming IPIs. in this note only IPIs for scan-fix are considered. these are processed directly out of the acquisition loop via the monitor ipi interrupt handler. there will be other IPI cases defined later, which will change this no doubt...

```
while (try to get root lock) == didn't {enable external interrupts; disable them}
```

4.12 Incoming IPI handler

Two cases:

- *drive all readers/scanners out of a particular ledger/page, typically so the ledger can be reset. nothing to do, just respond; the response implies that the receiver will re-acquire its ledger information before proceeding*
- *when a ledger is forgotten (eg, guest OS discards the associated code page), this requests that the receiver's TLB have the associated PTE flushed; response implies that the flush is complete and that the receiver will re-acquire any ledger information before proceeding. note that the details of what must be flushed will depend on allocation strategies for ledger structures, and needs to include any items where the va-to-pa relationship may change.*

Either of these implies the need to re-evaluate the 'current' guest OS page's ledger. If the interrupt is handled from monitor's get_root loop, this can be skipped (based on the CPL for the interrupt frame) since the monitor routine will eventually do a scan_return. If guest execution is interrupted, this routine itself needs to do a scan_return.

4.13 shuffle *fiddle-d-diddle to make iTLB present but dTLB absent*

this is a page fault; every page that has a ledger is actual-absent but os-present

```
get target addr from frame cr3
if there's no target ledger then this is not a shuffle page fault;
    other monitor causes TBS (eg, read from emulated memory-mapped IO), or
    jump to guest os page fault
assert(frame cpl not ring zero)
assert(os_present)
lock scanlock
if bump flag is reset
    find a new shuffle location
    scan from beginning of page for the first unscanned byte, or the first scanned instruction which is two bytes
    or more; if there aren't any in the whole page, we're doomed
    save address in aux.pbump
    set the bump flag in the flagblock for aux.pbump location
if byte at aux.pbump is unscanned  single byte case
    patch a return at aux.pbump
    mark actual pte valid
    flush TLB
    call the return via his mapping  loads code TLB but (we presume) not data TLB
    mark actual PTE not valid
    put original byte back
    unlock scanlock
    return
the two byte case
patch a bump and return at aux.pbump
mark actual pte valid
flush TLB
call the return via his mapping  loads code TLB but (we presume) not data TLB
mark actual PTE not valid
put original two bytes back
unlock scanlock
return
```

Supervisor Network Architecture

1. Introduction

The purpose of this document is to describe the network architecture for the Supervisor System.

The goals of the network architecture are

- To provide basic network connectivity to the individual worlds
- To protect the individual worlds from hardware faults or changes
- To provide redundancy
- To provide for transparent addition or removal of hardware
- To scale proportionally to the number of network cards and worlds.

For the duration of this document, the assumption is that the worlds will all be running a version of Windows that supports the NDIS miniport driver (ie, Windows 2K/XP or Windows CE). This is because the NDIS miniport driver is source code compatible across XP or CE.

This document provides an overview of the components and their interaction with each other. A detailed functional design document based on this document will follow.

2. The Components

The networking solution for Supervisor will consist a NDIS driver running in each OS and one or more LAN managers.

2.1 The NDIS driver

The NDIS driver will run on each OS that is being hosted by Supervisor. This will be a NIC miniport driver. This driver will not have direct access to the network card. When the OS (or any other application running in it) tries to send messages across the network, the NDIS library posts packets for transmission and informs the NDIS driver. The NDIS driver will then inform the LAN manager of the memory location (the physical address) of the packets. The manager will then transmit this packet over the NIC.

The NDIS driver will queue a set of buffers in to receive messages. The LAN manager will fill these buffers and mark them. The NDIS driver will then report to the NDIS library that a packet was received. After the library has copied out the data, the buffers will be put back in the queue.

2.2 The LAN manager

The LAN manager will be an EFI application. It will be the only component with access to the NIC. The manager will handle the transmission and reception of packets over the

NIC. If the message is for another world in the Supervised system, the manager will redirect the packet.

Conceptually, the manager will maintain a circular queue of outgoing packets. The manager will drain this queue when it sends the packets over the NIC. The queue will be filled when the NDIS drivers in the individual worlds try to send out packets. The individual elements of the queues will contain the physical addresses of the packets.

3. Communication between the LAN manager and the NDIS drivers

The problem of communication between the LAN manager and the NDIS drivers can be divided into two sub-problems: Discovery and Operation. Discovery refers to how the NDIS drivers and the LAN manager are going to be made aware of each other. Operation recovers to the actual process of sending and receiving data.

3.1 Operation

In normal operation, the communication will happen through send and receive queues. Both of these will be on memory that is shared between the driver and LAN manager.

The NDIS driver will enqueue a set of buffers to the receive queue. When a packet comes in for the world (from the NIC or from another world in the system), the LAN manager will copy the data in to the buffers, and set a shared counter, indicating the number of buffers received.

The driver will view this counter periodically to check for messages. If any messages have been received, the driver will inform the NDIS library, and once the library is done with the buffers, enqueue them back into the receive queue and reset the counter appropriately. Access to the counter must be synchronized, since both the LAN manager and the driver have access to it.

When the NDIS driver receives a packet from the NDIS library for sending, it will add the physical address of the packet into the send queue. The LAN manager will check the head of the send queues of the worlds in rotation. If any world has a packet to send, it will send the message out to the appropriate destination (either copy it out to a receive buffer in another world or inform the NIC of the packets physical address). After the packet has been transmitted (or an error generated), the LAN manager will update a flag to indicate that the packet has been sent.

It can be seen that the heads of the send queues form the circular send queue for the LAN manager.

The driver will poll the send queue periodically and take appropriate actions (clean-up where required, report errors etc).

It is advantageous for the LAN manager to poll because once there are a certain number of worlds running, the probability that there will be a message for transit at any point of time is quite high. In such a situation, it makes more sense to poll, rather than to give up the processor periodically and waste even more time getting swapped in and out of the processor. Having the LAN manager poll the worlds will also reduce the number of synchronization problems we are likely to have in the LAN manager.

Directing an interrupt at a particular world is likely to be a messy operation, for instance, we do not know if the world is currently executing, or is sleeping. Unless we can find a way of efficiently causing an interrupt to be raised at a specific world, I believe we will be better off having the worlds periodically poll the send and receive queues.

3.2 Discovery

Two things need to happen in the discovery process. One, the LAN manager has to be made aware of all the worlds it is supposed to manage. Two, the worlds have to be made aware of the LAN manager (Technically, it is only the NDIS drivers that are aware of a LAN manager).

The means to achieve this will be the Send and Receive queues.

When a world is brought up, the Thin Blue Line will allocate some memory for the queues and give it to the monitor for the world. When the NDIS driver comes up, it will make a call to the monitor, get this address and map it to the OSes virtual memory space.

The Thin Blue Line will give this same address to the LAN manager. The LAN manager can also generate a MAC address for the world at this point. Now, the LAN manager has some means of associating a queue with a world.

The Thin Blue Line will also maintain a record of the queues associated with a world.

If a LAN manager goes down and is restarted, or if a new LAN manager is brought in to share the load, all that needs be done is for the Thin Blue Line to give it the list of queues that it is required to handle.

This provides us with a clear separation between the LAN manager and the drivers. The drivers need not know which LAN manager is servicing them. This means that the LAN managers can be added or removed from the system transparently.

This separation also makes it easy to move the LAN manager functionality on to a smart card.

4. Prototyping and further work

For the prototype, the worlds will run Windows CE. This in itself should not affect us since the NDIS miniport driver is code compatible across CE and XP. For the prototype, the LAN manager will implement the basic functionality for communication described above. Redundancy and dynamic addition or removal of LAN managers can be added later.

Once the prototype is completed, the following directions can be explored

- Redundancy and dynamic addition/removal of LAN manager
- Efficient algorithms for polling the worlds and prioritising worlds.
- Optimizing inter-world communication
- Prioritizing the scheduling of worlds based on received messages
- Optimizing polling times for the worlds and the LAN manager for efficient operation.

0116.Doc

1 Introduction

The Extensible Firmware Interface (EFI) provides architecture for next generation system firmware.

1.1 Supervisor

Supervisor is platform firmware that defines multiple logical(virtual)-partitions within one physical-partition. The physical-partition of interest is ES7000 CMP partition and follow-on products.

1.2 EFI

EFI goals and requirements are compatible with hypervisor needs. EFI provides structure for hypervisor firmware implementation.

This brief EFI summary is extracted from the EFI 1.10 (.95) specification.

The EFI specification is primarily intended for the next generation of IA-32 and Itanium™-based computers. Thus, the specification is applicable to a full range of hardware platforms from mobile systems to servers. The specification provides a core set of services along with a selection of protocol interfaces. The selection of protocol interfaces can evolve over time to be optimized for various platform market segments. At the same time the specification allows *maximum extensibility and customization* abilities for OEMs to allow differentiation. In this, the purpose of EFI is to define an evolutionary path from the traditional "PC-AT†"-style boot world into a legacy-API free environment.

EFI Goals: Coherent, scalable platform environment; Abstraction of the OS from the firmware; Reasonable device abstraction free of legacy interfaces; Abstraction of Option ROMs from the firmware; Architecturally shareable system partition.

EFI Requirements: Evolutionary, not revolutionary; Compatibility by design; Simplifies addition of OS-neutral platform value-add; Built on existing investment.

EFI Attributes: Reuse of existing table-based interfaces (i.e. ACPI); System partition; Boot services; Runtime services; EFI Driver model

Implemented in C; Defines calling high level calling conventions; EFI driver model, bus/binding protocols. Three types of loadable modules: EFI Applications, EFI Drivers (Boot service/Runtime), EFI OS Loader
Boot only and runtime services provided to drivers and applications. When an OSLoader calls ExitBootServices, memory for boot only drivers is reclaimed.

2 Concepts

This section starts with a quick high level summary of EFI concepts. This is included here as the remainder of this document builds on these concepts.
This section finishes by introducing supervisor specific concepts.

2.1 EFI Runtime Environment

2.1.1 EFI Images are PE32+

EFI Images are a class of files defined by EFI that contain executable code. The most distinguishing feature of EFI Images is that the first set of bytes in the EFI Image file contains an image header that defines the encoding of the executable image. EFI uses a subset of the PE32+ image format with a modified header signature. The modification to signature value in the PE32+ image is done to distinguish EFI images from normal PE32 executables. The "+" addition to PE32 provides the 64 bit relocation fix-up extensions to standard PE32 format.

2.1.2 EFI Image Types

Applications and OS Loaders (which are a special case of application)
Drivers (Boot only or runtime)

For images with the EFI image signature, the *Subsystem* values in the PE image header are defined below. The major differences between image types are the memory type that the firmware will load the image into, and the action taken when the image's entry point exits or returns. An application image is always unloaded when control is returned from the image's entry point. A driver image is only unloaded if control is passed back with an EFI error code.

```
// PE32+ Subsystem type for EFI images
#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION 10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER 11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER 12
```

2.1.3 Environment Variables

EFI system configuration stored as a collection of environment variables. One important sets is used to install drivers (DriverOrder and Driver####). A second important set is used to define selectable boot environments (BootOrder and Boot####).

Portions of the supervisor implementation can be in zero or more drivers that provide runtime services to the supervisor EFI application. In normal operation the supervisor application would be the default boot environment.

2.1.4 Boot Manager

The boot manager is provided by the system core firmware. Supervisor can provide an enhanced boot manager to simplify the user interface.
Assumes normal boot of supervisor, other options provided for diagnostic purposes...

Supervisor.efi defined as the default boot option.

2.2 EFI Development Environment

Windows 2000 or later
Visual Studio 6.0 with Service Pack (SP4 or SP5)
VC++ Processor Pack.
MASM 6.11 (6.13)
EFI 1.1 (1.10.14.60) EDK (EFI Driver Kit)

2.3 EFI Component Architecture

Handles and Protocols are key abstractions used by the EFI architecture. A handle provides the means to reference modules and devices in the firmware system. A protocol is similar to a C++ abstract interface though defined to allow implementation in C.

2.3.1 Abstract Hardware

EFI uses method-based access to hardware instances. This allows drivers to provide implementations of abstract hardware interfaces. This minimizes coupling of layered drivers to hardware details. (For example EFI_PCI_IO_PROTOCOL has abstract concepts of IO ports, memory and configuration space.)

2.3.2 Handles

Every component and device is represented by an EFI handle. The EFI system maintains the runtime database of handles. This provides a standard mechanism for drivers to discover other components and services in the firmware system. The handles are also used to maintain driver dependencies and reference counts.

2.3.3 Protocols

A driver module entry point registers the protocols implemented by the driver. The protocols are associated with a handle that represents the driver.
A protocol instance is similar to an object instance in that instance data is associated with the protocol instance.

2.3.4 Services

The EFI system firmware core provides common services that are utilized by firmware drivers and applications. The services are divided into boot services and runtime services. Boot services available until EFI OS loader calls exit boot services.

Runtime services remain available to operating system. Typically requires cooperation by calling from boot processor with compatible addressing mode.

2.3.5 EFI System Table

The entry point for an EFI image receives a pointer to the EFI system table and a reference to the image handle.
The system table has references to boot and runtime services, interface references and handles to console in/out/err, and system configuration tables.

2.3.6 Device Paths

The core firmware and bus drivers are responsible for mapping drivers to devices. Bus and platform override mechanisms allow flexibility without changes to the core firmware.

2.3.7 EFI Runtime Library

The EDK provides a standard runtime library that allows common code sequences to be reused. The library includes routines for: initialization, linked lists, strings, memory, text IO, math, spin locks, handles, file IO, device paths, PCI 'macros', and several other miscellaneous functions.

2.4 EFI Driver Model

The EFI_DRIVER_BINDING_PROTOCOL is the basis for the EFI Driver Model.

2.5 EFI Protocol Categories

These standard protocols are defined in the EFI specification.

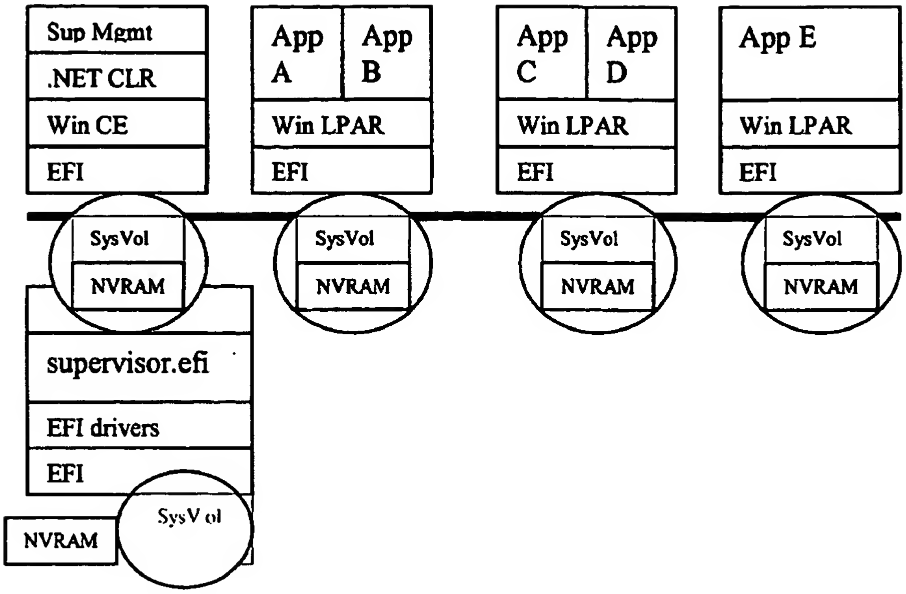
Console,	Text and graphics console
Bootable Image	Bootable Image
PCI BUS	PCI IO
SCSI Bus	SCSI Driver Model
USB	USB Driver Model
Network	Simple bootable network support
Debugger	Debugging port support
Compression	Standard compression and decompression
Device I/O	Generic device I/O

2.6 Supervisor Protocols

Custom protocols defined for manipulating logical partitions.

Partition	Partition Control
Partition Policy	Partition Policy
System Volumes	Partition Volume I/O

3 Architecture



3.1 *EFI Supervisor*

Physical machine partition
Owns physical memory map.
Can use 40bit PAE mode to manage memory map.
Creates 4GB or <=64GB PAE mode virtual partitions

Within each logical / virtual machine partition, 'OS' runs at ring 2.
Environment
Hardware / software
Applications run normally at ring 3.

3.2 *EFI loads supervisor components*

3.2.1 Supervisor EFI boot services drivers

Standard EFI drivers for standard hardware
Custom drivers with standard and custom interfaces as needed to boot supervisor

References added to Driver####, DriverOrder variables

3.2.2 Supervisor EFI runtim services drivers

Custom drivers with standard and custom interfaces as needed by supervisor runtime.
(supCPU.efi, supMem.efi)
Reference added to Driver####, DriverOrder variables
Supcpu.efi, manage additional cpu resourses
Supmem.efi manage memory map.

3.2.3 Supervisor.efi

EFI 'application' acts as OS Loader and calls ExitBootServices
Manages other processors and creates virtual efi environments. These environments map UGA console to window. [Console Manager provide remote access directly to virtual partition UGA console windows.]

Reference added to Boot####, BootOrder variables

3.2.4 Processor allocation

4 D v lopment Plan

4.1 *Initially create single processor virtual partitions.*

4.2 *EFI as VM monitor framework*

4.2.1 Native EFI IA32 implementation in virtual-partitions

4.2.2 SupervisorDisk.efi

4.3

4.4 *EFI x*

5 Issues

5.1 *Windows NTVDM*

BIOS dependencies
Initially restricted if necessary

5.2 *Windows WOW*

WOW environment machine and BIOS dependencies
Initially restrict WOW.

6 Tasks

6.1 *Supervisor EFI Driver*

6.2 *EFI32 Application*

6.3 *EFI friendly windows LPAR*

6.4 *EFI friendly user-mode-linux*

7 Integration Plan

7.1 *EFI on EFI*

Efi32.efi
Is an efi application that acts like OS loader. Simply maps [all] services 1-to-1 from stacked partition to the lower partition.
Run EFI shell in nested environment.

7.2 Multiple single processor EFI on EFI

Run EFI shell in each nested environment.

Docum nt Distribution List

	Name	Location		Name	Location
R	Alan Grub	Malven	A	Tim Case	Malvern
A	Bruce Vessey	Malvern			
R	John Landis	Malvern			

Legend: A - Approver
R - Required Reviewer
O - Optional Reviewer

_______ Approved Without Change

_______ Approved With Indicated Changes

_______ Not Approved

____
Signature of Approver / Date

_______ Comments

_______ No Comments

____
Signature of Reviewer / Date

Comments due by: 09/30/02

Notice : The printed version of this document may not be current.
Verify the version date against the on-line file in the Design Review System DataBase.

Abstract:

This is a draft of a white paper or proposal or functional design document for a CMP supervisor.

Disclaimer:

The primary purpose of this current document is as a working document for gathering, organizing, and developing the design of the various aspects of a supervisor for CMP based hardware.

The contents of this document may be useful input for formatting as either a proposal or as a functional design specification and one or more detailed design specifications using the appropriate EDL document templates.

UNISYS

Supervisor

Proposal Document

Date: 08/23/02

00000000-000
Version A

Approval Status: Being Written

Name	Role	Location	Mail Stop	Phone
John Landis	Author	Malvem	A29F	(610) 648-2497
Alan Grub	Collaborator	Malvem		(610) 648-xxxx
Jim Hunter	Collaborator	Malvem		(610) 648-xxxx
Bruce Vessey	Project Manager	Malvem	<MS>	(610) 648-xxxx
Tim Case	Director	Malvem	<MS>	(610) 648-xxxx
<Name>	Program Manager		<MS>	(xxx) xxx-xxxx

This page contains hidden text, which includes the Document Info Table and the macros used to update the table, restore bookmarks in the table, and remove hidden blue text when the document is complete. To view hidden text in Microsoft Windows, turn on the option to view hidden text.

Table of Contents:

1. Document Control

This document was generated using the PPG Template Generator, 3490 3880, revision J.

1.1. Change History

Began 5/10/2002 as private notes of John Landis for supervisor.

First shared with Alan G. Jim H. on 5/20/2002.

Major sections poured into proposal template starting on 8/20/2002.

1.2. Related Documents

Virtual machines are not a new idea, but until the recent resurgence, this has received little industry attention, especially for IA32 architecture computers.

TODO: Edit this section for readability and to focus on relevant concepts.

1.2.1. VM/360

Concepts fallen out of favor...

However, distributed application protocols have addressed many of the (usability) issues.

1.2.2. Bochs

This is an example of a cross-architecture virtual machine. Complete IA32 emulation in software can be ported to 'any' computer architecture.

Virtual IA32 - i.e. runs IA32 on other machine architectures (for example MAC). Relevant since Virtual devices are shared with Plex86 project.

1.2.3. Plex86

Formerly known as FreeMWare.

Plex86.org

Sourceforge open source project

"Running multiple operating systems concurrently on an IA32 PC using virtualization techniques", Kevin Lawton, 11/29/1999

Effort subsidized until 2001 by Mandrake Linux distribution.

1.2.4. Analysis of Secure Virtual Machine Monitor

"Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor" John Scott Robin, U.S. Air Force (scott_robin_@hotmail.com); Cynthia E. Irvine, Haval Postgraduate School, (irvine@cs.nps.navy.mil)

1.2.5. DISCO

Disco: Running Commodity Operating Systems on Scalable Multiprocessors

Stanford University

1.2.6. Cellular Disco

Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors

Extends DISCO by exploring fault tolerance of careful resource allocation…

Stanford University

1.2.7. VMware

Commercial product with three editions: VMware workstation; GSX; ESX. The ESX (Enterprise Server version) provides its own 'host' OS environment.

1.2.8. Other implementations

SWSoft virtuozzo

Connectix Virtual PC www.connectix.com

Innotek® Virtual PC for OS/2

1.2.9. Denali

"Denali: Lightweight Virtual Machines for Distributed and Networked Applications"

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble

The University of Washington

Para-virtualization

http://denali.cs.washington.edu

1.2.10. Intel

IA-32 Intel Architecture Software Developer's Manual Volumes 1-3:

IA32 Vol 1 Basic Architecture 24547006.pdf

IA32 Vol 2 Instruction Set Ref 24547106.pdf

IA32 Vol 3 System Programming Guide 24547206.pdf

Chapter 3, Protected-Mode Memory Management

Chapter 4, Protection

Chapter 10 Memory Cache Control

1.2.11. CMP

Unisys CMP

IPL Synopsis, TVP; "Synopsis (NEW and IMPROVED, includes card and software details)"

1.3. Document Cross Reference

2. Introduction

This document is loosely formatted as combination proposal and preliminary functional design. (The current plan is to reformat appropriate material with a standard proposal template.)

2.1. Purpose

Explore possibility of developing supervisor layer between high-end server hardware and 'commodity' commercial off the shelf server operating systems. Windows Advanced Server and Data Center Edition are of particular interest but other OS possibilities are not precluded.

Examine Cost/Benefit issues.

Cost: developer resource, elapsed time, risk, interdependencies

Benefit: Features, value, faster, better, safer

2.2. Scope

A supervisor layer between computer hardware and the operating system provides virtual partitions for multiple OS instances.

Hardware partitions require OS support to migrate resources between partitions on demand.

The supervisor can load balance the resources of the hardware (partition) among the virtual partitions. In particular, CPU and memory usage, as well as storage and network bandwidth, can by dynamically adjusted between the virtual partitions. There is also the possibility of transparently sharing physical memory pages between the virtual partitions.

This layer manages multiple OS system software environments with different versions and configurations to provide for more effective server consolidation than is possible using currently available operating systems and applications.

The .NET Framework built into Windows .NET Server (and subsequent versions) with support for managed code and side-by-side assemblies provides an increasingly capable platform for server consolidation.

2.3. Key Points

3. R quirem nts Analysis

<value added section>

This section is not the source of the requirements but rather explores the requirements to ground the remainder of the document on requirements.

Disclaimer: Don't get excited. Not all of these are expected to be actual requirements. It is expected that a 'line will be drawn' for each of these, and in some cases, multiple lines where a multiple release strategy makes sense. Certain items that are expected to 'fall below the line' are included anyway to help flesh out the other requirements. Eventually those items will be identified as non-requirements.

TODO: normalize implementation phase numbers across subsections rather than current relative phase numbers.

TODO: Include requirement identifiers where/when possible to reference the source of these requirements.

TODO: move non-requirements detail into subsequent sections of this document

3.1. Server Consolidation

A high-end server strategy competitively supports large-scale applications (like the database tier for example.) Server consolidation, which is a different aspect of a server scale up strategy, allows competitive hosting of many smaller applications.

Management of multiple virtual partitions on a single large-scale server is a solution for many of the limiting factors of large-scale consolidation. These limiting factors include: operating system scalability limits, applications with different OS requirements, application scalability, application single configuration assumptions, and application coexistence issues.

3.1.1. OS Scalability Limits

Resource limits within operating systems can limit application scalability. For example, the infamous limited system paged pool of the Windows NT family can limit application scalability. Allowing multiple instances of an application to run with different OS instances can avoid these limits.

3.1.2. OS Version Dependencies

Applications currently often have (conflicting) operating system version dependencies. This can be the limiting factor that prevents applications from running on the same OS instance. Multiple virtual partitions remove these constraints by allowing multiple versions of an OS, and also different operating systems to run in the same physical partition.

3.1.3. Application Scalability Limits

For some applications, scaling is either not implemented or was deemed 'too' difficult or unnecessary. For these non-scalable application architectures, multiple virtual partitions provide a mechanism to run multiple instances within a physical partition.

3.1.4. Application Configure-ability Limits

This is distinct from the previous situation, in that it is not uncommon for applications to assume only a single instance runs on a given operating system instance. Inflexible application configuration contexts may require different partitions to achieve desired configuration flexibility.

This can also be combined with poor application scalability characteristics that result in the need for multiple operating systems instances to achieve the desired scale, even though the application uses only a fraction of the system resources.

3.1.5. Application Coexistence Issues

If application A does not coexist with application B, the applications can run in different virtual partitions.

Windows .NET Server has made significant advances in this area with the introduction of side-by-side assemblies. This generally addresses the infamous "DLL Hell" problems. However, existing applications do not automatically benefit.

3.1.6. Application Coexistence Policies

Some applications have support policies that preclude or limit the ability to share an operating system instance with other applications. Even when all of the technical issues that limit coexistence are addressed, non-technical issues often remain.

3.2. Resource Allocation

Physical partition resources are allocated (sublet) to virtual partitions. It is possible to dynamically adjust (certain) allocations even when the virtual partitions are running.

Current operating systems typically do not support dynamic partitioning. For these systems, adequate resources must be allocated to a virtual partition before the partition is started. However, even for these systems, allocation of certain resources can be adjusted while the virtual partition is running. For example, the partition's share of the processor time or IO bandwidth can be adjusted independently of the operating system. Other resource allocations like the number of processors or the amount of memory can be adjusted while the virtual partition is stopped or hibernating.

Using virtual server clusters of two or more virtual partitions, resource adjustments without service outages are possible by stopping and restarting the virtual cluster nodes one at a time.

3.2.1. Virtual Partition

When a virtual partition is created, resource allocation constraints (or ranges) are defined. When a virtual partition is destroyed, any persistent resource allocations (e.g. disk volumes) should be reclaimed after (optional) scrubbing or destruction.

Between creation and destruction, all partitions support the stopped and running states. Some virtual partitions can support standby and hibernate states. These are similar but not identical to physical server power states. In this context, standby allows CPU resources to be reclaimed for use by other virtual partitions, and hibernate also allows memory resources to be reclaimed.

Virtual partitions cannot be started that would jeopardize resource commitments to running virtual partitions. However, virtual partition attributes may include 'batch' and related policy hints that allow resources to be reclaimed from running virtual partitions. The state of lowest priority virtual partitions can by changed to standby, hibernate or stopped as resource demands dictate and policy allows.

Virtual partition constraints provide allocation ranges and optional capabilities. Violation of these constraints is prevented to avoid operating system failures that would otherwise occur when the virtual hardware varies outside the tolerance of the operating system.

Virtual partition policy determines the priority of resources need relative to other virtual partitions. To simplify specification of virtual partition constraints and policy, predefined partition profiles should be available that map to known operating systems and usage scenarios.

TODO: move attribute list:

Stopping the physical partition host stops or hibernates all running virtual partitions.

virtual partition states: Stopped, Running, Standby, Hibernating

Virtual partition attributes define support for standby and hibernate states.

Implementation Phases:
1: Stopped and running only supported states
2: Standby state also supported.
3: Hibernate state also supported. Migration of virtual partitions between physical partitions supported.

3.2.2. Processor Allocation

Minimum allocation 5% of 1 processor to a virtual partition.
Allocation 5-90% of 1-32 processors to a virtual partition.

The percentage of processor allocation (5-90%) is dynamically adjustable for each virtual partition. The number of assigned processors is typically constant while a virtual partition is running. When the partition is stopped (or hibernated), the number of assigned processors can be modified, if allowed by the partition policy. The partition profile defines limits to the allowed changes. This allows the number of processors to be limited to the number supported and licensed by the virtual partition operating system.

A given virtual partition is either single processor or multi-processor. The multi-processor type allows the number of processors to be from 1 or 2 to 32. A related virtual partition attribute allows dynamic add/remove of processors.

Assuming partition profiles are available (see below for details on usability and simplicity enterprise attributes), the profiles for Windows 2000 and Windows .NET Server would not allow processors to be added or removed except when the virtual partition is stopped.

TODO: move this to resource sharing and/or to overview/definition.

Given current lack of operating system support for processor hot add/remove, it might be helpful if there were some sort of processor lease back mechanism. This would allocate an excess number of processors that could be dedicated to some sort of high priority idle task so that they would not need to be actually assigned to a physical processor. (They would virtually run.) If needed they could be 'dynamically' added to the partition without restart. Profile may allow virtual real time application to be installed on the virtual partition, which by running with processor affinity at real time priority is able to lend one or more processors back. This mechanism might be used to allow processor capacity to be reserved in advance and dynamically changed without needing to stop or hibernate the virtual partition.

(Processor imposes limit of 640 virtual partitions using 5% of 1 processor of a 32x physical partition.)

Implementation Phases:
1: Fractional allocation not supported. Virtual partitions of 1 to 4 processors supported. Changes to number of processors allowed only when the virtual partition is stopped.
2: Fractional allocation supported. Fractions in 5% granularity of 1-32 processors. Fraction can be

dynamically modified when virtual partition is running. Number of processors modified only when virtual partition stopped.
3: Hot add/remove of processors supported for capable virtual partitions

3.2.3. Memory Allocation

Each virtual partition is allocated a current, minimum and maximum amount of memory (e.g. 256MB, 128MB-4GB)

4GB allocation

>4GB allocation (implies PAE mode for the virtual partition)

Performance implications of >4GB allocation are TBD.

A virtual partition attribute enables "Hot Add" memory support. This can be utilized by Windows .NET Server.

A separate virtual partition attribute enables "Hot Remove" memory support. Windows does not yet support hot remove. Other operating systems may also support this capability.

For any operating system, memory size can be changed (reduced to minimum) when the virtual partition is stopped. Limited changes may be valid when the partition is hibernating.

(Memory imposes limit of 512 virtual partitions using 128MB of a 64GB physical partition.)

Implementation Phases:
0: Virtual partitions run only in non-PAE mode. (4GB max memory)
1: Memory changes only when partition is stopped. (64GB max memory)
2: Hot add memory supported
3: Hot remove memory supported

3.2.4. IO Slot Allocation

PCI slots can be individually assigned to a virtual partition. (Limited number of non-shared bus-master adapters can be supported.) Shared slots/interfaces are assigned to special IO virtual partitions.

For operating systems that support 'plug and play' and PCI hot plug, slot assignments can be changed when the partition is active. All partitions allow assignments to be changed when the partition is stopped.

The expectation is that IO slots are typically allocated only to the IO virtual partitions.

Implementation Phases:
1: virtual compatibility board for each virtual partition. No additional slot assignments
3: IO slots can be assigned to stopped virtual partitions
4: Hot add/remove of slots to virtual partitions

3.2.5. IO Throughput Allocation

Priority policy for storage and network bandwidth.

Pending shared storage requests are processed based on the configured bandwidth policy. If a given virtual partition is allocated 50%, then when this partition has pending requests, on average, 1 of 2 storage requests will be selected from this partition. (TBD Share is may be calculated by request only or it may include data transfer size.)

Virtual network packets are dropped if a low priority partition is flooded with packets. (This simulates behavior of standard network routers.)

Implementation Phases:
1: static policy
2: dynamic policy changes

3.2.6. Dynamic Physical Partition

It is important to note that the supervisor can support dynamic resource allocation to its physical partition. This includes hot add/remove of processors, memory, and IO capability.

This may be important to support desired reliability by allowing replacement of field replaceable units while the virtual partitions continue to operate.

There are valid reasons why customers would want to separate mission critical, production, and non-production virtual partitions into different physical partitions. The ability to dynamically adjust resources between the physical partitions would allow more effective resource utilization.

Implementation Phases:
0: Physical partition must be stopped to change resources
2: Memory resource changes while physical partition running. Available to stopped virtual partitions.
3: Processor resource changes
4: IO resource changes

3.3. Resource Sharing

? Is this helpful as a distinct section, or would it be better to merge into virtual partition attributes or server consolidation section?

Sharing the physical partition resources among multiple virtual partitions provides opportunities for additional consolidation through over-committing. However if all virtual partitions should demand resources at the same time, policies must dictate which virtual partitions starve.

Note the possibility of migrating virtual partitions to other physical partitions on the same or other physical servers, in the same or other networked data centers.

Batch partitions are those that are performing non-time critical work and can 'volunteer' to be suspended, hibernated, or stopped as needed. Note that completion deadlines typically exist even though real-time response is not required.

3.3.1. Time Warps

Various time warps may be detectable from code running within the virtual partitions. This is unavoidable due to the bursty nature of processor scheduling. Operating systems with extensive power management capabilities are expected to be less sensitive since sharing a processor is similar to power management clock slowdown or duty cycling.

Time of day

Virtual partition elapsed time.

Time between virtual timer ticks.

3.3.2. Time Accounting

Operating environments may provide skewed timing when running in a virtual partition. The monitor itself can/should provide rudimentary billing. This can be derived automatically or manually from the virtual partition scheduling policies. This could allow billing information generated by virtual partitions to be scaled to provide useful information.

3.3.3. Over-committed Processors

Successful over-commitment of processor resources requires detection of the idle loops in the virtual partitions. Operating systems that support power management and hot add/remove of processors are likely to provide the necessary capabilities. (In data center context, given increasing power demands of processors, operating systems can be expected to become more proactive in reducing power requirements by halting unneeded processors.)

Operating Systems that are hyper-threading aware, and use the HALT instruction as recommended by Intel, would enable this detection. Other mechanisms are possible (and riskier) but not necessary.

Putting 'batch' virtual machines on standby would provide a low latency mechanism to reclaim processor resources for reassignment to restarted idle processors.

Implementation Phases:
0: Over-commitment prevented
1: Over-commitment of non-batch virtual partitions prevented. Batch virtual partitions are swapped/hibernated as necessary.
2: Idle processor detection only through halt. Idle time lent to batch virtual machines to increase the processor utilization of the physical partition.
3: Over-commitment prevented only at the data center level, virtual partitions migrated between physical partitions as necessary.

3.3.4. Over-committed Memory

Over-committed memory allows for more granular resource sharing. However it also carries a greater risk of page thrashing.

A memory-HAL would be helpful to allow operating systems to trade free pages with other partitions as the operating load changes.

Backing read-only pages of 'batch' virtual partitions to a hibernate-file would provide a low latency mechanism to reclaim memory resources when all virtual partitions require memory at the same time.

Implementation Phases:
0: Over-commitment prevented
1: Over-commitment of non-batch virtual partitions prevented. Batch virtual partitions are suspended (put on standby) as necessary.
2: Available pages detected and lent to batch virtual machines to increase memory utilization.
3: Over-commitment prevented only at the data center level, virtual partitions migrated between physical partitions as necessary.

3.3.5. Shared physical Memory

This approach can allow virtualization memory overhead to be reclaimed. (Otherwise some memory overhead per virtual machine is necessary when calculating maximum number of concurrent virtual machines.)

Even without over-committed memory, sharing physical pages between virtual partitions may provide a performance advantage if it can be implemented such that at least some levels of the memory cache are shared.

Implementation Phases:
0: No pages shared

2: Memory pages copied via network interface shared (copy on write)
3: Memory pages sourced from storage shared (global buffer cache maintained)

3.4. Operating System Aspects

TODO: edit section for readability

Given current trends (processor performance and memory cost, customer education), operating system acknowledgement of virtual machines would seem to be inevitable. In the longer term, this may allow optimizations as industry standard abstraction layers and interfaces are developed.

In the meantime, managing the details of virtual partition compatibility of particular operating systems is important.

3.4.1. OS support in virtual partitions

Which operating systems do virtual partition environments support?

Similar target as current CMP servers.

Predominate usage of Windows server family, but one or more open versions of Unix need to be supported. (Linux or OpenUnix)

Must it be possible to install and boot, shrink-wrap versions of an OS, or is customization with drivers and abstraction layers acceptable?

This relates to time to market with support for new versions of supported OSes and support a greater number of OSes.

Guest OS kits ready for use with virtual drivers pre-packaged. Minimal time to market for new kits. (The supported OS versions are those with available kits.)

Implementation Phases:
0: Windows .NET Server (Standard)
1: Windows .NET Server (Enterprise, Data Center);
2: Windows 2000
2: OpenUnix, Linux

3.4.2. Library OS

Lightweight OS (e.g. Exokernel approach) option for applications targeted only for a VM environment. Given library availability, a simple direct port (relink only) of applications that use (some subset of) Win32 API and/or GNU/Posix API.

Implementation Phases:
4: Library provided for small WIN32 subset
5: Library provided for GNU/Posix

3.4.3. OS Innovation

By providing low cost virtual servers, additional opportunity for operating system innovation is possible. In a physical server environment, applications are constrained by 'peer pressure' to adopt high volume or high probability operating environments. In a virtual server environment, operating systems choices can be made by individual applications and can be changed more easily. This environment emphasizes interoperation with network and data center protocols.

3.5. Hardware Aspects

TODO: edit section for readability

3.5.1. Hybrid Partitions

Combination of physical and virtual partitions

Virtual partitions run within physical partitions

Perhaps certain OS stages only in physical partition

TODO: Investigate possibility of sub-partitions that leverage CMP hardware to reuse industry standard IO hardware with minimum overhead.

3.5.2. Server Hardware Innovation

A software layer tightly coupled with the hardware allows for software collaboration with server system hardware design. To achieve this with commodity operating systems requires a separate firmware or 'supervisor' layer more capable than a typical SAL (system abstraction layer). This new layer theoretically allows faster/fewer hardware cycles by providing an opportunity to adapt or correct minor hardware deficiencies in software without multi-vendor dependencies and resultant schedules.

In this scenario, a supervisor may provide hardware fixes to a single 32x 'virtual' machine image running on 32x server hardware.

3.5.3. Packaging Density

Achieve enterprise packaging-density higher than blade approaches, since each virtual machine (blade) does not need unique system back plane or other infrastructure.

Supervisor manages physical CPUs so that CPU limit of commodity OS does not constrain the number of CPUs in a hardware partition.

A physical hardware partition with 128 IA32 processors could be managed by a supervisor to run 4 to 512 virtual partitions.

Hardware partitions allow system image with more CPU resources than can be utilized by any commodity OS.

3.5.4. Cells

Potentially marginally higher cost of entry (minimum 8-way to play) though low cost starter cells that are only partially populated could be provided. Half or Quarter cells could provide lower cost of entry for development and prototype purposes. (Starter cells would not need to be upgradeable to full cells.)

Multiple cells combined for maximum scale up, also allows more resource sharing with virtual partitions.

3.5.5. Virtual Appliance Servers

Canned boot drive images can support virtual appliance servers.

In effect these are pre-configured OS instances.

Create additional instances (based on licenses) and upgrading multiple instances in controlled fashion are key value adds of virtual appliances.

Implementation Phases:
3: File Server Appliance
4: Print Server Appliance

3.5.6. Virtual Blades

Buzz surrounding blade implementations ignores virtual partition approach. Virtual blades can be at least as manageable as actual blades.

3.5.7. Manage Blade Infrastructure

Customer real need is to manage availability of application workloads. Deployment of applications on physical or virtual blades could be transparent.

Industry standard hardware advances can be harnessed by architecting the virtual partition management capability such that it can be applied to racks of multiprocessor blades. (Possible hedge against lack of hardware availability.)

Implementation Phases:
3: Cell management infrastructure includes drivers to manage industry standard blades

3.6. Virtual Partition Attributes

TODO: Move details from this section to "Supervisor Virtualization" section, to focus this subsection on technical requirements.

3.6.1. Virtual System

Virtual time may interfere with system time accounting policies. (It may be possible to accommodate with CPU speed identification fibs.)

Accurate time accounting within the virtual partitions is a non-requirement for initial implementation.

Virtual ACPI used to describe the virtual ports provided to each virtual partition.

Limit number of traditional devices/busses.

It would be convenient if only virtual EFI firmware (and not virtual BIOS) was necessary. Windows IA32 does not yet support EFI firmware, but support is a possibility in the Longhorn timeframe. Linux x86 can boot now using only EFI (session at spring 2002 IDF)

In any event, virtual BIOS is needed to support Windows 2000 Server and Windows .NET Server.

3.6.2. Virtual CPU

Virtual partitions configured with 1-N processors, where N approaches the number of processors in the physical partition.

Physical CPUs (maximum granularity in FRU bundles) can be added and removed from physical partition.

(Is support of more than one physical partitions required?)

Gang scheduling required so spin locks work as expected. This requires a relatively course grained scheduling algorithm.

What CPU identification and features do the virtual partitions expose?

Is it dependent on the configuration and/or resource allocation for the virtual partition?

3.6.3. Virtual Memory

Virtual Physical memory, physical-to-machine map

Trap OS requests to update TLB (Translation-Look-aside-Buffer)

Page map for each virtual machine

Supervisor TLB cache

Machine memory map - entry for each real machine memory page. (list of VM/VA, ref to replicated copies)

Physical partitions greater than 4GB expected to be typical. This implies that VMM would typically run in PAE mode.

Virtual partitions less then 4GB expected to be typical. This implies that VM would not typically run in PAE mode. This suggests that the VMM would translate between non-PAE page tables of the VM into PAE mode page tables of the monitor. Support for PAE mode VM could also be provided.

3.6.4. Virtual IO

Limit support to generic network and storage if possible with standard virtual devices for storage HBA and Ethernet NIC.

Is custom IO hardware required in any target markets (e.g. TELCO)?

PCI passthrough allows IO adapters to be dedicated to single virtual partitions.

PCI passthrough for dedicated hardware could also support custom IO hardware.

Generic virtual IO at PCI level is useful to limit interdependencies. Once a virtual partition is booted, the network and storage stack uses some optimized path. This is useful to simulate the variety of 'standard PC' devices that are necessary to boot. (Windows 2000 requires a shorter list than Windows NT 4.0)

3.6.4.1. Console

Bootstrap scenario for installation and network initialization.

Virtual Console Manager IPL2 with virtual firmware running outside the virtual partition.

Long term EFI/UGA is appropriate.

'Performance' drivers targeted at specific operating system versions.

Windows .NET Server 'Headless feature' requires only virtual serial port.

Industry/OS standard Remote Desktop Connection available once VM (network stack) is up and running.

3.6.4.2. N twork

Each virtual partition must have one or more MAC address assigned.

3.6.4.3. Storage

No need to virtualize directly attached disk volumes. LUNs of directly attached disks do not need to be shared across virtual partitions by the supervisor.

Call OS boot volume and application data volumes all be required to be supplied via storage servers (e.g. SRM)

Advances in network-attached storage may eventually make separate virtualization of storage redundant, though it may survive as a storage protocol offload mechanism. (Similar to current iSCSI HBAs that offload performance sensitive portions of the TCP stack to the 'network' interface.)

3.7. Enterprise Attributes

These are often known as the 'abilities'. Most of these are required for any opportunity in the enterprise space.

3.7.1. Reliability

It is assumed that a fault in a virtual partition is isolated from other virtual partitions.

In addition, supervisor should carefully assign physical resources to virtual partitions in an attempt to limit the hardware failure exposure of individual virtual partitions.

It should be possible to group virtual partitions by support categories. (See "New Services Packaging for ES7000/200": Mission Critical (Revenue); Production; Non-production(develop/test).

3.7.2. Maintainability

Supervisor can migrate virtual partitions away from selected field replaceable hardware units (FRU) to allow replacement without interruption of active virtual partitions.

(This requires physical partitions to contain more physical CPUs than the largest virtual partition.)

3.7.3. Manageability

Programmable (scripting) interfaces to manage virtual partitions.

Create/destroy virtual partitions.

Start/stop (standby/hibernate) virtual partitions.

Query and Modify virtual partition resource allocation policy.

Support optional multi-level change logs for persistent storage to allow virtual partitions to 'time travel'.

3.7.4. Scalability

This is key requirement. Without significant scalability, the value of a supervisor is severely limited.

Sentinel capabilities, including self-healing, should be applied to the virtual partitions.

3.7.5. Usability

Ability to migrate and maintain virtual systems should require minimal specialized training.

User Interface to management interfaces.

It's not tractable to manage partitions one attribute at a time.

Multiple profiles:

OS profile, attribute restrications for OS

Workload profile

Focus UI on application workloads.

Workload – super assembly manifest

Workflow rules for application mix staging from test to production

3.7.6. Flexibility

Hybrid partitions

3.7.7. Simplicity

Simplify, don't need many ways for same capability. One great way will do…

OS Profiles for virtual partition required capabilities.

4. Int rd p nd ncies

5. Functional Overvi w

6. Concepts

<value added section>

TODO: significant editing required of entire section

Virtual machines are not a new concept.

Processor architectures are unequal in their ability to enable reliable implementations of virtual machines. (The IA32 architecture is less capable than many others.)

6.1. CMP

Cellular Multi Processing provides uniform memory access (UMA) among all cells of system. (Contrast with ccNUMA.)

Virtual partitions would allow supervisor to manage ccNUMA like interconnection of multiple large CMP (UMA) clusters.

6.1.1.FRU

Field Replaceable Unit (FRU)

Supervisor can dynamically reassign physical resources at granularity of field replaceable units so that none of the virtual partitions in the physical partition are using the unit. Replacing the unit than requires coordination only with the supervisor.

This also allows dynamically adding physical resources to a physical partition which the supervisor than allocates among the virtual partitions. Excess resources can also be removed in a similar fashion.

6.1.2.Physical Partitions

Server control software allows physical partitions to be dynamically created. These are similar to a limited number of virtual server blades.

(Unfortunately, partition features are oriented around currently active partitions. Managing alternate configurations for an intended use require customer naming-conventions.)

6.1.3.Multiple Domains

Multiple power domains allow additional reliability. Unnecessary spanning of domains by partitions impacts the reliability of the partition. (Spanning can increase performance at cost of reliability.)

6.1.4.CMP Server

A given server supports one or more physical partitions.

Allocation boundaries provided by hardware.

Although SAN (storage/server/system area network) technology allows partitions to migrate between servers, current management software does not facilitate this capability.

6.1.5. Multiple racks in multiple data centers

Cluster implementations, and large-scale data centers, involve multiple systems in a data center.

Architecture should not preclude load balancing and fail-over between physical partitions in different CMP systems or data centers.

6.2. Virtual Partition Architecture

Adopt virtual machine concepts to increase the capabilities of servers does not require abandonment of physical partitions. Virtual partitions managed within a physical partition remove granularity limits. Valid reasons exist to contain virtual partitions with different categories in different physical partitions. However, this does reduce the hardware requirements for physical partitions. In appropriate situations, a server with a single static physical partition may be adequate.

Virtual cells of variable size.

6.2.1.4.2.1. Enhance COTS at lower cost

Resource balancing code added with out major changes to existing Operating Systems.

(Note that Windows .NET Server does include NUMA capabilities.)

6.2.2.4.2.2. Fractal OS

Fractal implementation of resource management: CPU, memory, ...

Another layer of a true enterprise OS.

Leveraging HAL/SAL and pluggable drivers minimizes virtualization effort and/or virtualization overhead.

Given current trends (processor performance and memory cost), operating system acknowledgement of virtual machines is inevitable. As this occurs, industry standard abstraction layers and interfaces will be developed.

This allows traditional operating systems to focus on high-level abstractions for sharing resources between relatively tightly coupled applications. Virtual Machine Monitors implement low-level (machine) abstractions for sharing resources between operating environments.

6.2.3. No OS imposed CPU per system limit

OS is no longer the limiting factor in number of CPU supported per system/partition.

Windows NT (2000,.NET) has architectural limit of 32 CPUs for IA32. (APIs contain word sized processor affinity masks.)

(On 32x XEON MP system, 32 physical processors and 32 logical processors can be utilized.)

{If 4 cells can be interconnected, what if each cell contained more than 8 CPUs? (This already somewhat possible with current generation of hyper-threading processors.)

Since the supervisor can allocate CPUs between the VM instances, the limiting factor on the CPU/cell ratio becomes memory bandwidth from cache to shared memory rather than OS architecture.}

6.2.4.RDMA

Remote DMA protocols to optimize distributed performance. IO operations for (and between) virtual partitions can be partially completed (the memory transfer phase) without needing to immediately schedule the partition.

This capability is expected to be leveraged by storage and networking to optimize performance.

(This protocol can be used between virtual partitions.)

6.2.5.Services as Applications

Storage and Network as applications hosted in a virtual partition.

Protocol offload to storage and network application processors.

Fractal variant of micro-kernel approach (See the "GNU Hurd" operating system for an interesting example of this.)

Running storage and network application in a virtual partition leverages multiple processor host environment to minimize monitor size and thus increase reliability.

Policy service reduces amount of management code resident in the monitor. A custom adapter allows the monitor to deliver 'critical' events to the policy service.

Console service provides secure remote access. The full resource demands of the console service only need to be incurred when a remote console is active.

6.2.6.Virtual Blades

Virtual partitions provide the benefits of blades but at potential lower cost in large scale implementations. (Smaller blade incremental unit cost not relevant in large installations with many blades where blades can be expected to be purchased in multiples of at least 4 or 8.)

Virtual blades allow 'independent systems' to be packaged within a 'cell'. Each blade contains CPU, Memory, Storage, and Network access.

Blades are about system packaging density and unit of incremental upgrade. Multiple blades plug into shared chassis which can be partially populated. (http://www.dell.com/us/en/biz/topics/power_ps1q02-blades.htm)

Cells (CPU/Memory integrated with chassis) with virtual blades, can provide (even greater) packing density but with a larger incremental upgrade cost. This may be more appropriate than physical blades for large enterprises.

<Resource Sharing>

<Windows Technology>

6.3. Virtualization Technical Details

6.3.1. IA32 no explicit virtualization support

19 sensitive instructions do not generate traps. Extraordinary means are required to prevent OS code from discovering that it is running in a virtualized environment. Depending on the nature of discovery, it could result in incorrect operation.

6.3.2. Gang Scheduling

Virtual CPUs of the virtual partitions are (must be) gang scheduled due to typical spin lock behavior. A VCPU that owns spin locks must execute so that it can release them ASAP.

6.3.3. Virtual Ethernet

VMM provides MAC address for each VM and a virtual switch. Traffic destined outside of the VMM domain is routed through a physical NIC.

6.4. Possible Approaches

6.4.1. Virtual Infiniband

Leverage MS Infiniband as rack infrastructure.

VMM implements virtual Infiniband HCA (Host Channel Adapter) rather than virtual Ethernet.

Ehternet and Storage are standard implementations that run over virtual HCA.

6.4.2. Para-Virtual IA32'

{This is a (independent) variant of Denali para-virtualization approach.}

An industry standard virtual-IA32 definition would allow Microsoft to "Port NT" to a 'virtual IA32'

Assumption that distribution of ~17 non-virtualizable sensitive kernel instructions are predominately in HAL and possibly in kernel. The 'port would entail HAL and possibly some components in kernel. References if any in other executive components should be 'fixed'.

'Porting' Linux to this v86 architecture should be limited to the 'architecture' source branch.

7. Vendor Opportunities

<value added section>

This is currently just a brainstorm list of potential fixes vendors might make that would improve the performance and reliability of virtualization.

7.1. Intel

7.1.1.IA32 Xeon

Supervisor does not need to run on commodity desktop hardware or even commodity server hardware. As a consequence, supervisor can be designed to leverage capabilities of the latest Xeon silicon. This suggests there are potential opportunities to exploit new capabilities or virtualization tweaks in the latest server CPUs or microcode.

Fostor, Galiton

7.1.2.Virtualized IA32

This mess is all Intel's fault in that though the 80386 provided virtual 8086 and 80286 support, it did not provide virtual 80386 support. Self-virtualization support has not yet been added to IA32 architecture.

Other architectures (including 8086 and 80286) explicitly support virtualization.

A machine specific register setting could cause the infamous 17 instructions to generate traps. An architectural machine specific register would be convenient but not necessary. No need for 'using' limited supervisor status register bits for this capability.

(It's worth noting that 'simple' virtualization does not imply efficient virtualization support. For example, a poorly designed protection scheme may have unacceptable trap overhead. That's where the system abstractions come into play.)

7.1.3.Fast Call

Built-in OS Fast call mechanism (SYSENTER/SYSEXIT instructions) designed for Ring 3 applications calling Ring 0 operating system. When a supervisor layer present, this mechanism is not optimal.

7.1.4.Multi-level OS

Other architectures, i.e MIP R4000, differentiate kernel and supervisor modes. This allows 'supervisor' to run as machine kernel that handles traps as necessary from operating system running in supervisor mode.

IA32 rings potentially useful except that ring level is not transparent (some of 17 instructions expose CPL without protection), and paging hardware supports only two level (user and supervisor) protection.

7.1.5.IDT Protection

Ability to capture selected IDT vectors (machine check for example) would allow software resolution of server specific issues...

Generating special virtualization traps to IDT, GDT would be useful virtualization subset.

7.1.6.More VMs than physical space

PTE virtualization support?

Maintaining accessed and dirty bits in virtual PTEs is a potential performance problem.

Simplify multiple-level paging to avoid pitfalls of supervisor page-in for OS page-out.

7.1.7. Intel IA64

IA64 provides a virtual IA32 user mode (ring 3) but not a virtual IA32 supervisor mode. This allows Windows to provide WOW64 environment, but doesn't provide hardware support for virtual machine environments.

In all the fervor over virtual IA32 machines on IA32 hardware, wouldn't preclude possibility of virtual IA32 on IA64 hardware.

Now is the time to be looking at what is required to support virtual IA64 machines on IA64 hardware.

7.2. Microsoft

7.2.1. Intel recommended hyper-threading optimizations

See IA32 Vol3 section 7.6

Use 'Pause' instruction in spin lock loops

Halt Idle processors

Guidelines for scheduling threads

Eliminate execution-based timing loops (use local APIC timer or the time stamp counter)

Place locks and semaphores in separate aligned 128-byte blocks of memory

Use CPID to determine number of logical CPUs per package.

Theoretically, VMM could provide virtual partitions with 1 physical package with 4 to 8 logical processors and schedule them to logical processors across physical packages as necessary.

7.2.2. Virtual IA32 friendly VAL/SAL/HAL

Define a Virtual Abstraction Layer (VAL). The VAL may be a superset of companion to physical SAL interface.

A Microsoft supported Virtual IA32 HAL with system vendor provided SAL similar to IA64 architecture. Key hooks for virtualization should be provided in this SAL interface. Only the virtual; IA32 HAL defines VAL interface.

Implementation of many of the remainder of these items would be simplified by the existence of a VAL layer.

7.2.3. Tim r Overflow Count

Virtual timer hardware should define timer overflow count register, incremented for periodic timers on each reload of the current count from initial count register. Cleared when driver reads the overflow count and acknowledges the interrupt. This prevents timer skew by allowing timer driver to be aware of all timer ticks without the need to handle an interrupt for each tick.

Without this capability, the VMM must generate a virtual interrupt for each count and decrement the overflow count for each acknowledged interrupt.

7.2.4. Sleep Interval

Avoid gang scheduling of virtual partition to deliver timer tick interrupt that needs to be simply counted. It is preferable

OS typically assumes overhead of timer tick interrupt is 'minimal' since it is not aware of virtual partition scheduling latencies.

7.2.5. Processor Affinity

When rescheduling, awareness of cache hierarchy (which includes lowest level physical package cache) and cache for nodes of 4 physical processors.

7.2.6. Network Timers

Network stack (TCP/IP) implementations typically assume delivering periodic timer to network stack is an inexpensive operation. In a virtual partition environment, this requires waking up an otherwise 'idle' partition to deliver individual ticks.

A more virtual partition friendly approach would utilize event based code and timestamps at critical processing points. In some situations, comparing timestamps can replace use of periodic timers.

7.2.7. Halt Idle Processors

By halting CPUs in the idle loop, detection of idle processors is more practical. Processors known to be idle to not need to be scheduled with the virtual partitions CPU gang. A self-tuning system can further use statistics on number and time of idle processors to initiate add/remove of virtual partition processors.

7.2.8. Hot Add/Remove Processors

Explicit support of processor hot add/remove provides an additional resource allocation adjustment mechanism. Additional processor resources can be added to non-clustered partitions and restart is not required for clustered partitions.

7.2.9. Virtual HCA miniport

Microsoft is developing Infiniband stack that includes a HCA miniport. Microsoft had planned to own all miniport implementation (as they do for USB for example) Depending on the definition of HCA hardware interface, there may be a performance win in defining a virtual HCA miniport. This would allow future versions of Windows NT to install/boot/run without the need for supervisor provided drivers.

The latest is that MS is making Infiniband stack available for some sort of vendor redistribution. Integrating a virtual miniport may be possible without cooperation.

7.3. Infiniband Trad Association

Investigate whether Host Channel Adapter (HCA) specification, assuming a standard interface is defined, accommodates virtual implementations.

7.4. Unisys

IA32 not limited 32X

Virtual IO Channels

Cache

8. Functional Description

9. Architecture

TODO: Put pictures here.

TODO: picture of 'thin blue line'

TODO: picture of IO processor (in VP or in addin IO card (aka Intellifibre)

TODO: picture of schedule map

10. Components

<Value added section, may move this to companion functional design specification>

10.1.'Objects'

This summary does not attempt to design these objects. It does however attempt to design the object relationships.

10.1.1.Physical Partition

This maintains and manages the state of the physical partition.

The state is subdivided (via objects) into physical CPU, Memory, IO resources.

The state includes reservation/assignment information of resources to a virtual partition. If a virtual partition fails, the resources can be reclaimed without trusting state within the virtual partition.

If this data is compromised, all virtual partitions are at risk.

Maintains a reference to the Idle Virtual Partition, which by definition has a virtual CPU for each physical processor.

Maintains reference to the 'current' Policy Virtual Partition.

Maintains reference 'list' to the IO Virtual Partitions.

Maintains reference 'list' to all Virtual Partitions.

10.1.1.1.Physical CPU

The physical-CPU object tracks which virtual partition is currently running on its CPU. The CPU state (register contents etc) are saved within the virtual CPU objects. Delayed saves of special purpose registers (e.g. floating point, MMX) are possible if the physical CPU object tracks multiple virtual CPU objects.

10.1.1.2.CPU Schedul

The CPU schedule object is a circular schedule of CPU time quantum's for each of the physical CPUs. The policy virtual machine is responsible for most changes to the schedule. When a virtual machine is destroyed, the monitor (physical partition) replaces all references to the decommissioned partition to the idle virtual machine.

TODO: Add description of schedule earlier in the doc and include pictures.

The schedule table has 20 rows of N CPUS. There is one row for each 5% increment of CPU time. Each virtual partition assigned to contiguous rows and columns to maximize affinity. The policy virtual partition is responsible for 'best fit'.

10.1.1.3.Physical Memory

The physical memory object tracks the physical memory ranges assigned to the partition.

Owns data structure that maps physical pages to virtual machines. When virtual partition removed from page, if no other virtual partitions are mapped, the physical page is reclaimed.

Initial implementation exclusively assigns (1) range to each virtual partition.

10.1.1.4.Physical IO

Bus

PCI adapter

Ownership of each physical IO resource to exactly one virtual partition.

Map for owner of each resource.

10.1.1.5.Virtual channel

One virtual channel for each connection between virtual partition and IO virtual partition.

10.1.2.Virtual Partition

This maintains and manages the state of one virtual partition.

The state is subdivided into virtual CPU, Memory, IO resources.

If this data is compromised, the virtual partition may fail or misbehave, but state of other virtual partitions cannot be affected.

Maintains reference to 1-N virtual CPU, including identification of the boot processor.

Maintains reference to virtual memory,

Maintains reference to (or copy of) virtual partition policy.

10.1.2.1.Virtual CPU

Contains CPU state save area.

When running, 'reference' to physical CPU object.

State for various clocks like the virtual RDTSC value.

Boolean for virtual boot processor.

Machine specific registers for the 'implemented' virtual processor(s)

10.1.2.2.Virtual Memory

Manages physical to virtual physical to virtual mappings.

Manages free pages.

10.1.2.3.Virtual (Shared) IO

Timers

10.1.2.4.Dedicated IO

10.1.2.5.Virtual Channel Endpoint

10.1.3.Boot

10.2.Machine Monitor

The machine monitor owns the physical partition context and is responsible to manage the virtual partition contexts.

The implementation is not monolithic but is composed of modules that manage well defined contexts. Modules implementing methods of physical variant of various objects are what some would think of as monitor components. Modules: CPU, Memory, Interrupts, IO,

This is in effect an extension of the system firmware. The Extensible Firmware Interface (EFI) model used to provide structure to the monitor components.

10.2.1.Monitor

Trap and interrupt handlers.

IDT for physical partition

?IO Monitor, VC endpoint, emulation of PCI resource

10.2.2.Virtual Partition Monitor

Create and destroy virtual partitions.

Track location of virtual GDT, IDT, page directory, page tables

Manage allocation and contents of machine partition GDT, IDT, page directory and page tables.

Possibility of multiple partition monitor implementations.

A 'para-virtualization' implementation would be much smaller and potentially more reliable. This might be useful for policy partition implementation and maybe IO partition.

10.2.3.Virtual Partition Scheduling

Simultaneously schedule as many virtual partitions as possible. Uses gang scheduling of all non-halted processors in the virtual partition.

10.2.4.Virtual Channel Manager

Creates and reclaims virtual channels that connect virtual machines or virtual machines to IO emulation layer.

The virtual partition endpoint is either terminated by a driver in the virtual partition or by the IO emulation layer for the virtual partition.

The other end is terminated by a driver in the IO Manager virtual partition, or by a machine monitor IO driver.

10.2.5.Virtual Channel Context

Context for one virtual partition.

Future implementation of shared page mappings for copy-on-write sharing between virtual partitions

10.2.6.Boot

Boot strategy for the monitor depends on the hosting environment.

BIOS/Firmware boot mechanisms one alternative.

Host OS another alternative.

10.3.Processor Virtualization

Because of the scope of this implementation, this component is listed as a sibling of monitor…

Manages controlled execution (to fix 17 infamous instructions) and also replaces performance sensitive instruction sequences.

Expect multiple implementations and/or iterations to obtain information while allowing integration of other supervisor components.

10.3.1.Trace Implementation

(When performance is of no concern, the trace interrupt can provide controlled execution of virtual ring 0 code.)

This is maintained so it can be used for problem isolation.

10.3.2.Production Implementation

This is a hard problem when performance is a concern.

Scan and fix

Emulate instructions

10.3.3.Loader

Where does this fit in? If processor virtualization has access to native code files and can map pages to code files, opportunities for optimizations.

Maps code sections to descriptors of known code

10.4.Virtualization Driver Model

Two major abstractions: Legacy Busses (like PCI ;-) and virtual channel endpoints.

The virtual partitions contain a virtualization driver model (VDM.) These drivers allow adapts to developing technology with ripple effects to other major components. The major abstraction is a virtual channel endpoint. The ideal guest OS provides its own native drivers for the channel endpoint. This would be conceivable if/when Infiniband HCA standards mature.

10.4.1.EFI

EFI firmware model basis for VDM interfaces. Interfaces defined by EFI will be adopted wherever possible. Note that EFI does not attempt to replace ACPI but rather complement it.

Since support for guest OS without EFI is almost certainly required, EFI implementation is actually wrapper over x86 BIOS. Small changes to sample implementation expected.

10.4.2.BIOS

This legacy component is supported until all supported guest OS are EFI compatible (Windows Blackcomb timeframe…)

This is expected to be minimal 'port' of existing ES7000 system BIOS.

10.4.3.ACPI

ACPI BIOS modification to build virtual ACPI tables for each virtual partition.

10.4.4.CMOS

Non-volatile memory for firmware.

Interacts with policy storage virtual partition for persistence of CMOS and updates.

10.4.5.PCI Emulation

Emulates arbitrary PCI devices. Maintain maps of ports, memory mapped ranges, interrupts for mapping to individual virtualization drivers.

Two coupled components. Bus abstraction combined with generic PCI device. (Is a generic PCI device useful?)

10.4.6.Timer

The timer is relatively tightly coupled with virtual partition scheduling. This driver is a subcomponent of the scheduling and timing services subsystem.

10.4.7.Channel

The generic channel endpoint is tightly coupled with virtual IO. This driver is a subcomponent of the virtual IO subsystem.

Infiniband HCA details. Basic mechanism expected to be provided by underlying virtual channel. Ideally, this driver only is involved in IO setup operations rather than mainline IO path.

PCI Express model provided as alternate endpoint driver. Guest OS drivers of PCI express cards all use generic PCI express virtual driver.

USB as a 'channel'.

10.4.8.Storage

ATA with DMA,

Serial ATA

10.4.9.Network

NIC

This component not required if virtual console provides NIC that can be used for maintenance, and guest OS provides NIC driver for a supported channel.

10.4.10.Console

Virtual IPL2 multifunction PCI card. Host for VGA/UGA BIOS. Host for USB BIOS.

10.4.11.PCI Passthru

Minimizes overhead of mapping virtual partition IO accesses directly through to unshared hardware.

10.5.IO Applications

These components may be hosted by either the monitor or by an IO virtual partition. Moving these applications to an IO virtual partition provides increased robustness at some cost to performance. A separate partition allows failure of this software to be recovered like a temporary network failure.

10.5.1.Infiniband Switch

When/If Infiniband materializes in the server space, providing Host Channel Adapter (HCA) interface

10.5.2.Ethernet Switch

A native implementation of Ethernet switch.

10.6.IO Drivers

These components may be hosted by either the monitor or by an IO virtual partition. Since the monitor needs timer services for scheduling the virtual partitions, at least a portion of a timer 'driver' will need to be host directly by the monitor.

10.6.1.Timer

A native implementation of timer handlers. Provides interrupt handlers for the timer hardware assigned to the physical partition. Distributes timer events to the virtual partitions.

10.6.2.Channel Adapter

A native implementation for Infiniband HCA terminates generic virtual channel endpoints with targets outside of the physical partition.

It may be that Infiniband switch terminates all endpoints, and switches traffic destined outside the physical partition through this driver.

10.6.3.Storage

A native implementation for Intellifibre HBA terminates virtual channel storage endpoints.

This is optional. Use of other storage hardware is provided by an instance of the IO Virtual Partition.

10.6.4.Network

TODO: identify high performance NIC for native driver and qualification efforts.

This is optional. Use of other network hardware is provided by an instance of the IO Virtual Partition.

VIA model on Intellifibre hardware would obviate need for IO virtual partition.

10.7.Guest Virtual Partition

Drivers installed in virtual partition environment to reduce virtualization overhead. Driver constructs IO request packets and interacts with virtual channel queues with minimal interactions and context switches.

For operating environments that do not support high level abstractions for IO, or for performance sensitive IO, custom drivers provide means to introduce higher level abstraction.

The performance sensitive driver classes are expected to be storage and network. For highly interactive administration scenarios additional drivers for console may be desirable.

10.7.1.Console

Traditionally this would imply principly KVM.

(Think host components of IPL2 project.)

Video, preferably EFI UGA (could wrap VGA BIOS implementation) If VGA BIOS required, licensing of IPL2 implementation may be possible.

USB (keyboard, mouse, floppy, CDROM) USB aware OS provides drivers. Windows 2000 requires redistributing of RNDIS drivers if needed for access to 'virtual maintenance LAN'. The IPL2 package could be used for this.

Performance video drivers could be provided (The standard IPL2 drivers should work here. Some tweaks might be useful for timing issues that might occur.)

10.7.2.Channel

When/If Infiniband materializes in the server space, providing Host Channel Adapter (HCA) interface will allow Microsoft provided stack to obviate the need to provide custom drivers for the virtual partitions.

This requires the monitor to implement an Infiniband switch.

Now that MS is not including stack in .NET or Longhorn, there is opportunity to redistribute with supervisor. Tweaks to MS HCA miniport might be useful.

This component is the guest OS endpoint of a virtual channel. Infiniband provides opportunity to deliver industry standard version first time out. A proprietary implementation (i.e. IB like) is also a possibility. High performance storage and network drivers provide class specific mappings to the provided channels.

10.7.3.Storage

SCSI class. Terminates virtual channel storage endpoints, generates SRB IRPs

MS Infiniband implementation one option.

10.7.4.Network

NDIS 'application'. Terminates virtual channel network endpoints. Generate NDIS IRPs.

MS Infiniband implementation one option.

10.8.IO Virtual Partition

High throughput storage and network traffic. This is a 'slight' performance compromise for range of supported drivers.

(move to summary section) Range of implementation approaches:

- Embedded Windows Server
- User mode IO subsystem application emulate BSD/Linux driver models
- Infiniband switch model (hardware implementation)
- Intellifibre intelligent IO model

Rejected alternative:

Runs embedded version of Windows .NET Server in a virtual partition.

Uses PCI passthru mechanism to gain reuse of COTS drivers.

One custom driver per IO class, built to route IO from virtual channel endpoints.

Proposes alternative:

Single IO application linked with 'library OS' runs in the IO Virtual Partition at a single non-zero protection ring. This obviates the need for the monitor to control execution of the IO virtual partition.

An IO application manages BSD style driver modules. The application manages the 'bottom' end of the IO virtual channels. Based on resource allocation policy, routes requests from the channels to the driver modules.

The application has source modules that manage each of the supported IO classes.

10.8.1.Storage

SCSI class. Terminates multiple virtual channel storage endpoints to share HBA for SRB from multiple virtual partitions.

10.8.2.Network

NDIS 'application'. Terminates virtual channel network endpoints. Generate NDIS IRPs.

10.9.Console IO Virtual Partition

Virtual Console Manager card runs in lightweight virtual partition.

Interacts with Console VDM, and Console clients.

TBD: does each virtual partition get companion console virtual partition, or do all virtual partitions share one console virtual partition, or do virtual partitions in the same security context share a console virtual partition.

10.10.Policy Virtual Partition

One server (or server cluster) per physical partition.

Operation policy

Scripting environment

Virtual PCI Interface (including interrupt) for interaction with the Virtual Machine Monitor.

This policy server is responsible for optimization of VCPU scheduling.

The monitor schedules using simple algorithm against data (the CPU schedule) manipulated by this server. Changes to the schedule made via 'transactions', one approach is alternating schedules with monitor switch to the 'latest' at current table wraparound.

Watchdog timer used to restart this virtual partition in the event of failure.

Scheduling tables can be reconstructed if necessary.

.NET Framework environment for management environment.

The policy virtual partition could be used as the idle virtual partition. An explicit idle virtual partition might be created.

10.11.Policy Storage Virtual Partition

One virtual server cluster per server farm (or data center).

This server owns the storage volume (database) that contains the virtual partition configuration files. This allows virtual partitions to be transparently moved to other servers with access to same storage area network (SAN).

Likely protocol is HTTP (or WebDAV). Distributed Authoring and Versioning (CIFS one alternate protocol)

Web Services protocols used by policy virtual partition to obtain XML formatted configuration document.

10.12.Idle Virtual Partition

Contains virtual CPU objects for any CPUs that the policy virtual partition is not interested/capable of claiming.

11. Us r Interfaces

12. System Software Interfaces

13. Proposal

13.1.Proposal Summary

Requirement ID	Requirement	Commitment	Source	Proposal

13.2.Compatibility

13.2.1.Hardware Compatibility

13.2.2.Software Compatibility

13.3.Migration and Coexistence

13.4.Performance and Connectivity

13.5.Interoperability and Conformance

13.6.Reliability, Availability, and Serviceability

13.6.1.Reliability

13.6.2.Availability

13.6.3.Serviceability

13.7.Maintainability

13.8.Subcontractor Process Requirements

13.9.Internationalization

14. Installation and Release Packaging

15. Implementation Tools

16. Testing

17. Significantly Altered or Deleted Features

18. Issues and Risks

App ndix A. Definitions, Acronyms, and Abbreviations

IO Channel Driver

A supervisor component loaded by the IO Virtual Partition. It receives IO requests from other virtual partitions for shared resources (i.e. Storage or Network) and according to sharing policies, forwards the requests to an appropriate IO Device Driver for servicing.

IO Device Driver

A supervisor component loaded by the IO Virtual Partition. It receives IO requests from the IO Channel Drivers and delivers them for servicing to the shared IO hardware.

IO Virtual Partition

A system critical virtual partition that allows other virtual partitions to share IO resources of the physical partition.

OS Driver

OS Specific driver

OS Device Driver

OS specific driver for a particular device

Physical Partition

is a hardware partition in a physical machine. Supervisor divides physical partition resources among one or more virtual partitions.

Policy Assembly

A versioned and signed component loaded by the policy virtual partition that automates one or more aspects operation of the virtual partitions. Standard assemblies are provided with the system. Customers can provide additional assemblies.

Policy Virtual Partition

is a system critical virtual partition than manages the operation policy of all of the virtual partitions. Customer provided extensions can customize automation of scheduling adjustments.

Virtual Ethernet Switch

is provided by the Ethernet implementation of an IO Channel Driver that loops Ethernet packets back though virtual channels to appropriate virtual partition. Connection of the virtual switch to a physical NIC is optional. (Virtual Ethernet Switches can be created and virtual partitions can be explicitly configured to use one or more.)

Virtual Machine Monitor (VMM)

Monitors virtual partitions (machines) as necessary to protect them from each other.

Virtual Partition

one of the virtual machine environments. Classified as one of: System Critical, Mission Critical, Production, Non-production. System critical partitions

Virtual Partition Firmware

Firmware loaded within virtual partitions. Possible implementations are ACPI BIOS and EFI.

Virtual Partition Monitor (VPM)

Virtualization Driver

Driver that virtualizes a particular aspect of a virtual partition.

Appendix B. Alt rnativ s Consid red

Title:	Supervisor	00000000-000
Type:	Proposal Document	Version A
Owner:	Bruce Vessey	
Date:	08/23/02	Page 1 of 47

Document Distribution List

	Name	Location		Name	Location
R	Alan Grub	Malven	A	Tim Case	Malvern
A	Bruce Vessey	Malvern			
R	John Landis	Malvern			

Legend:			
	A	-	Approver
	R	-	Required Reviewer
	O	-	Optional Reviewer

________ Approved Without Change

________ Approved With Indicated Changes

________ Not Approved

____

Signature of Approver / Date

________ Comments

________ No Comments

____

Signature of Reviewer / Date

Comments due by: 09/30/02

Notice : The printed version of this document may not be current.
Verify the version date against the on-line file in the Design Review System DataBase.

Abstract:

This is a draft of a white paper or proposal or functional design document for a CMP supervisor.

Disclaimer:

The primary purpose of this current document is as a working document for gathering, organizing, and developing the design of the various aspects of a supervisor for CMP based hardware.

The contents of this document may be useful input for formatting as either a proposal or as a functional design specification and one or more detailed design specifications using the appropriate EDL document templates.

UNISYS

Supervisor

Task List

Date: 01/08/2002

00000000-000
Version A

Approval Status: Being Written

Name	Role	Location	Mail Stop	Phone
John Landis	Author	Malvern	A29F	(610) 648-2497
Alan Grub	Collaborator	Malvern		(610) 648-xxxx
Terry Powderly	Collaborator	Malvern		(610) 648-xxxx
Bruce Vessey	Project Manager	Malvern	<MS>	(610) 648-xxxx
Tim Case	Director	Malvern	<MS>	(610) 648-xxxx
<Name>	Program Manager		<MS>	(xxx) xxx-xxxx

This page contains hidden text, which includes the Document Info Table and the macros used to update the table, restore bookmarks in the table, and remove hidden blue text when the document is complete. To view hidden text in Microsoft Windows, turn on the option to view hidden text.

Tabl of Cont nts:

1. Docum nt Control

1.1. Change History

Derived from presentation prepared by TVP

2. Dev lopm nt Tasks

These development tasks are significant and useful subsets of supervisor functionality.

The goal is to develop these tasks independently on individual development/test PCs.

Intermediate outputs of these tasks are leveraged as tools in subsequent iterations of other of these tasks.

Key intermediate stages of the independent tasks used for integration are identified.

The proposed independent tasks are:

- Virtual Partition Firmware
- Thin Blue Line
- VLAN
- VDISK
- VCONSOLE
- ADMINISTRATION
- Windows CE v-x86 Kit

2.1. Virtual Partition Firmware

This is the firmware for the virtual server partitions.

2.1.1. Starting Point

EFI as a process started from EFI

Develop methods to
- start EFI context from EFI environment
- Define minimum set of IPC messages
- Based on EFI standard

2.1.2. Purpose

Develop (virtual) EFI firmware that runs in the virtual partitions.

2.1.3. Summary

It begins as a new EFI32 build target to EFI source tree. (BUILD\efi32; COREFW\fw\platform\BuildTip\efi32) This is similar to (derived from) NT32 build tip, but loaded as EFI application (efi32.efi) rather than as NT console application (nt32.exe).

Early iterations are very thin and delegate services to underlying 'standard' EFI implementation.

Morphs into supervisor firmware build tip (vm32) and split from early version when later iterations begin to depend on 'supervisor' specific services provided by the 'thin blue line'.

2.1.4. Interdependencies

Depends on EFI Specification, and Sample implementation.

Early intermediate iterations consumed by TBL prototypes, and later iterations use Thin Blue Line services.

2.1.5. Conjectures

2.1.6. Iteration Plan:

0: Hello.efi, LoadHello.efi

1: delegates all services to underlying EFI instance

11: provides minimal services (CONSOLE, LOADFILE) can run Hello
This is iteration useful for TBL testing.

12: Adds BLOCKIO services mapped to image file in underlying instance

2.2. Thin Blue Lin

This is the Virtual Server Backplane.

2.2.1. Starting Point

Transaction code to “the thin Blue line

•Develop transaction rules for communication with TBL data structure

•Define recovery policy

2.2.2. Purpose

Package Supervisor specific enhancements to standard EFI implementation that are loaded in hardware partition.

Provides very low level virtual cpu scheduling (i.e. switch to next software partition.)

Implements low-level resource allocation transactions (for memory, cpu, virtual channels).

It provides low-level mechanism to create/destroy and start/stop virtual/software partitions.

It manages EFI ‘system partition’ storage container (and NVRAM) for each software partition.

2.2.3. Summary

2.2.4. Interdependencies

Inserts ‘virtual partition firmware’ into newly created software partitions.

It manages virtual channel endpoints for network, disk, and console capabilities.

2.2.5. Conjectures

2.2.6. Iteration Plan:

0: create/destroy empty software partitions

1: allocate memory/cpu to software partitions (no virtual code executes)

2: inject ‘virtual partition firmware’ into software partition memory

10: Schedule one or two simple EFI32 environments with very limited services.

20: Virtual channel resources

2.3. vLAN

Virtual partitions connect to this high-speed Virtual Network. It includes (optional?) custom OS drivers.

2.3.1. Starting Point

EFI with vLAN driver•Using the UGA model develop
–XP driver code mods to NE2000 driver
–OS-EFI LAN driver
–Interface with IO-EFI with LAN switch

EFI with LAN switch•Develop IO-EFI interface to OS-EFI for LAN
•Develop LAN centric CACHE model
•Develop switch type management interface
–Support Dual paths for recovery
–Support multiple paths for performance
–Support IO priority/flow models
•Port 10/100/1000 driver for NIC
–Use EFI INTEL driver if suitable

2.3.2. Purpose

Provide high performance network access to each software partition.

Potentially share memory for data communicated between software partitions.

2.3.3. Summary

Provide custom network drivers for each supported guest operating system. For Windows systems the driver would be a NDIS miniport. A higher performance option could include additional components required to support SDP (Sockets Direct Protocol).

The custom network drivers are supervisor aware, and redirect network packets through a supervisor provided virtual channel endpoint.

2.3.4. Interdependencies

Virtual channel endpoints provided by TBL

2.3.5. Conjectures

Shared memory implementation as starting point.

EFI platform provides adequate programming flexibility and power for LAN manager implementation.

Win CE driver model is similar enough to Win XP so that custom drivers are same or very similar.

(Raj is currently investigating the driver model for CE.)

2.3.6.Iteration Plan:

2.3.7.Components

2.3.7.1.NDIS mini-port

NDIS miniport built with Windows DDK

MSDN: "Windows CE 3.0 Technical Articles" "NDIS Implementation in Microsoft Windows CE Platform Builder 3.0"

Preliminary Component Design from Raj:

- Create an NDIS mini-port driver that will fake a NIC on the VM.

- The NDIS library posts packets for transmission and informs the NDIS driver. The NDIS driver will then have to inform the LAN manager of the memory location of the packets. The LAN manager can then send it out to the NIC.

- When a packet is received, the LAN manager will have to inform the NDIS driver on the appropriate world. The NDIS driver can then copy the data out of the LAN driver's buffers to a local buffer and then inform the NDIS library.

- We can use some code from the NDIS driver written for shared memory.

- The Windows CE library supports the NDIS miniport format, so hopefully porting shouldn't be a problem.

2.3.7.2.Virtual Ethernet Server

Host for switch/bridge/router/firewall
(Guest OS host for network infrastructure: DHCP/DNS, run as instances of embedded server appliance.)

2.3.7.3.Switch

For environments where network packets don't leave virtual network, the virtual switch would not need to bridge to the hardware NIC.

2.3.7.4.Router

Explicit (IP) router component uses occasional packets at upper layers to encapsulate virtual MAC addresses within the virtual subnet that connects directly to the switch. The router connects the switch to the physical NIC. (Implements ARP and maybe RIP protototcol.) It could be tightly coupled with the switch to minimize queuing latencies for packets that traverse the hardware NIC.

2.3.7.5.Firewall

A NETFILTER like implementation could be embedded in the switch. Management of the filter rules could then in a separate virtual partition.

2.3.7.6.DHCP/DNS server

Instances of embedded server appliance (running in virtual partitions) leverage industry standard implementation of standard network protocols.

2.3.8.EFI-LAN Manager

EFI-LAN manager

Preliminary Component Design from Raj:

- Only component with direct access to the NIC.

- A circular queue of buffers to take in packets sent by indivdual NICs. As the NIC get a new packet, they will fill the queue up. The queue will be drained as the LAN manager sends these packets out into the NIC.

- As a new message comes in, the manager will inform the appropriate world's NDIS driver, and the packet can be copied out.

- The manager can implement a multi-level queue so that priority levels for VMs can be implemented.

- The manager will need to be designed in such a way that it can handle incoming and outgoing packets efficiently

2.4. vDISK

Virtual partitions connect to this high-speed Virtual SAN (Storage Area Network). It includes (optional?) custom OS drivers.

As IP Storage infrastructure matures, this capability can leverage the virtual networking.

2.4.1. Starting Point

2.4.2. Purpose

Provide high performance disk storage access to each software partition.

Potentially share memory for volumes shared between software partitions. This capability would be a shared disk buffer cache.

Maximize leverage of Storage Sentinel.

2.4.3. Summary

2.4.4. Interdependencies

2.4.5. Conjectures

Storage Sentinel (symmetric version) can directly manage the volumes for the virtual partitions.

The virtual disk (buffer cache) is the process that owns the HBA that is connected to Storage Sentinel Server.

2.4.6. Iteration Plan:

2.5. vConsol

This provides the boot consoles for the virtual server partitions. The client is the 'standard' Console Manager partition desktop.

Runtime consoles are expected to leverage networking and operating system specific remote consoles. (i.e. Windows Remote Desktop Connection using RDP.)

2.5.1.Starting Point

2.5.2.Purpose

Provide operator access to system boot console.

2.5.3.Summary

2.5.4.Interdependencies

2.5.5.Conjectures

2.5.6.Iteration Plan:

2.6. Administration

Administration and Management interfaces for the virtual/software partitions servers.

2.6.1. Starting Point

External Management

2.6.2. Purpose

Provide Programmable and User Interfaces for control of the virtual partitions.

Create/Start/Stop/Destroy.

Resource allocation assignment/limits for basic memory/cpu/disk/network resources.

Manage the container/contents of the virtual system partition volumes for the virtual EFI firmware.

2.6.3. Summary

Administration is provided by a process running in a virtual partition using supervisor interfaces to the TBL. This process hosts 'scriptable components' of the programmable interface and provides network access for standard and custom user interface clients.

2.6.4. Interdependencies

It requires the supervisor interface to the management primitives implemented by the TBL.

2.6.5. Conjectures

The administration process would run ASP.Net possibly running on Windows CE. This would provide both web UI and programmable web services.

2.6.6. Iteration Plan:

2.7. WinCE v-x86

Port Windows CE to virtual partition architecture.

2.7.1. Starting Point

Prototype guest operating system kit.

2.7.2. Purpose

Provide prototype guest operating system that supports Win32 API and standard Windows device driver mini-ports.

This is a simpler port with more available source than the more capable Windows XP, or Windows Server 2003.

2.7.3. Summary

Boot Windows CE using EFI

(Assume run Windows CE with no BIOS interfaces.)

2.7.4. Interdependencies

2.7.5. Conjectures

2.7.6. Iteration Plan:

2.8. LPAR HAL

This is embedded XP and/or embedded Server.

2.9. virtual loader

EFI loader for Windows (2003)

2.10.Hijack ring 0

2.11.XP at ring 1/2

EFI with ring 2 XP

- Dependant on starting XP
- Define requirements for limited ring 2 execution
  - HAL enough for demo environment?

App ndix A. D finitions, Acronyms, and Abbreviations

IO Channel Driver

A supervisor component loaded by the IO Virtual Partition. It receives IO requests from other virtual partitions for shared resources (i.e. Storage or Network) and according to sharing policies, forwards the requests to an appropriate IO Device Driver for servicing.

IO Device Driver

A supervisor component loaded by the IO Virtual Partition. It receives IO requests from the IO Channel Drivers and delivers them for servicing to the shared IO hardware.

IO Virtual Partition

A system critical virtual partition that allows other virtual partitions to share IO resources of the physical partition.

OS Driver

OS Specific driver

OS Device Driver

OS specific driver for a particular device

Physical Partition

is a hardware partition in a physical machine. Supervisor divides physical partition resources among one or more virtual partitions.

Policy Assembly

A versioned and signed component loaded by the policy virtual partition that automates one or more aspects operation of the virtual partitions. Standard assemblies are provided with the system. Customers can provide additional assemblies.

Policy Virtual Partition

is a system critical virtual partition than manages the operation policy of all of the virtual partitions. Customer provided extensions can customize automation of scheduling adjustments.

Virtual Ethernet Switch

is provided by the Ethernet implementation of an IO Channel Driver that loops Ethernet packets back though virtual channels to appropriate virtual partition. Connection of the virtual switch to a physical NIC is optional. (Virtual Ethernet Switches can be created and virtual partitions can be explicitly configured to use one or more.)

Virtual Machine Monitor (VMM)

Monitors virtual partitions (machines) as necessary to protect them from each other.

Virtual Partition

one of the virtual machine environments. Classified as one of: System Critical, Mission Critical, Production, Non-production. System critical partitions

Virtual Partition Firmware

Firmware loaded within virtual partitions. Possible implementations are ACPI BIOS and EFI.

Virtual Partition Monitor (VPM)

Virtualization Driver

Driver that virtualizes a particular aspect of a virtual partition.

14 February 2003 .... Larry Krablin

This is a (very) drafty draft of a document which enumerates the issues involved in getting guest software to behave acceptably under supervisor.

1 Simplifying Assumptions

There are a number of assumptions we can make for this project which somewhat simplify handling of the issues described in this paper. This section will contain an (eventually) exhausted list of them. For now, the following are noted:

- late model Pentium 4 and Xeon only
- explicitly Unisys qualified (maybe cooked) O/S and driver guests
- Unisys platforms and Unisys qualified devices only
- regardless of guest intent, operation will always be in protected mode, with paging enabled
- 32 bit mode only?

2 Sensitive Stuff

2.1 CLTS – Clear Task-Switched Flag in CR0

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. It forces a #NM exception to be raised for (almost) any FPU instruction until it is cleared. CLTS would typically be used in the handler for this exception.

Its use is a potential visibility violation because it alters hard machine state outside the control of the monitor. Its effect in practice is dubious, as task switching rarely occurs in Windows.

This item raises questions about general handling of the FPU, MMX, SSE, and SSE2 state. In a preemptive scheduling environment, FSAVE/FRSTORE have to be used, with appropriate care, when switching between virtual machines, unless those switches are done with formal task gate calls.

It is protected by raising a #GP(0) exception when CPL > 0.

Resolution:

- raw – just emulate on interrupt
- cooked – maybe eliminate?
- demo – emulate on interrupt

2.2 HLT – Halt

The HALT instruction causes the processor, or hyper-threaded logical processor to enter the halted state.

I don't know what the Supervisor response might be to the occurrence of HALT in guest software.

It is executable only in at CPL = 0; otherwise a #GP(0) exception is raised.

Resolution:

- raw – emulate on interrupt?
- cooked – translate/scan&fix to supervisor call
- demo – if it happens, emulate on interrupt

2.3 IN – Input from Port

IN uses I/O port addresses, which may not be actually present or directly available to the guest. It also is intended to interact with extra-processor hardware, which must be mediated by the supervisor.

If virtual (emulated) ports are provided by the virtual machine, the supervisor must provide the interpreted action. Otherwise, this is an error.

The appropriate combination of CPL > IOPL & permission bits in TSS will cause the generation of a #GP(0) exception.

Resolution:

- raw – emulate on interrupt
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, emulate on interrupt

2.4 INS/INSB/INSW/INSD – Input from Port to String

same as IN above

2.5 OUT – Output to Port

same as IN above

2.6 OUTS/OUTSB/OUTSW/OUTSD – Output String to Port

same as IN above

2.7 LGDT – Load Global Descriptor Table Register

The GDTR contains the pointer to the Global Descriptor Table, which is the fundamental data structure for both memory management and general protection. The source operand is a linear address and limit for the table.

Guest software cannot be allowed to change the table or its contents without intervention by the supervisor.

See further discussion in the Memory Management section below.

Execution of this instruction at CPL > 0 raises a #GP(0) exception.

Resolution:

- raw – emulate on interrupt

- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, emulate on interrupt

2.8 SGDT – Store Global Descriptor Table Register

SGDT returns the linear address and limit for the Global Descriptor Table, as contained in the GDTR.

Since the supervisor will maintain the actual descriptor tables, the results returned are likely to appear to be inconsistent with the setting the guest software thought it loaded into the GDTR. Some intervention must be made to return results that won't disturb the blissful ignorance of the guest.

See further discussion under Memory Management below.

There is no architectural protection for SGDT.

Resolution:

- raw – use scan & fix to replace with INT3 or other exception raising instruction
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, suffer the consequences (!)

2.9 LIDT – Load Interrupt Descriptor Table Register

The Interrupt Descriptor Table (IDT) is an array of up to 256 gate descriptors that serves as the dispatch table for interrupts of all kinds. LIDT uses a linear address and limit to load the IDTR.

The IDT is crucial to system control and any fiddling with it must be mediated by the supervisor.

See also discussion of LGDT and Memory Management.

Execution of this instruction at CPL > 0 raises a #GP(0) exception.

Resolution:

- raw – emulate on interrupt
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, emulate on interrupt

2.10 SIDT – Store Global Descriptor Table Register

SIDT returns the linear address and limit for the Interrupt Descriptor Table, as contained in the IDTR.

Since the supervisor will maintain the actual descriptor tables, the results returned are likely to appear to be inconsistent with the setting the guest software thought it loaded into the IDTR. Some intervention must be made to return results that won't disturb the blissful ignorance of the guest.

See further discussion under Memory Management below.

There is no architectural protection for IGDT.

Resolution:

- raw – use scan & fix to replace with INT3 or other exception raising instruction
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, suffer the consequences (!)

2.11 LLDT – Load Local Descriptor Table Register

The Local Descriptor Table (LDT) serves a purpose parallel to that of the GDT. LLDT uses a segment selector to a segment descriptor to be found in the GDT to load the LDT, rather than a linear address and limit.

Supervisor considerations are similar to those for LGDT.

Guest software of interest may not use LDT?

See further discussion under Memory Management.

Execution of this instruction at CPL > 0 raises a #GP(0) exception.

Resolution:

- raw – emulate on interrupt
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, emulate on interrupt

2.12 SLDT – Store Local Descriptor Table Register

SLDT returns the segment selector for the Local Descriptor Table, as contained in the LDTR.

Since the supervisor will maintain the actual descriptor tables, the results returned are likely to appear to be inconsistent with the setting the guest software thought it loaded into the LDTR. Some intervention must be made to return results that won't disturb the blissful ignorance of the guest.

See further discussion under Memory Management below.

There is no architectural protection for SLDT.

Resolution:

- raw – use scan & fix to replace with INT3 or other exception raising instruction

- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, suffer the consequences (!)

2.13 LMSW – Load Machine Status Word

LMSW has some of the functionality of MOV CR0. See discussion there.

LMSW is also a legacy that, with a little luck, doesn't appear much in modern software (yeah, right).

Execution of this instruction at CPL > 0 raises a #GP(0) exception.

Resolution:

- raw – emulate on interrupt
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, emulate on interrupt

2.14 SMSW – Store Machine Status Word

SMSW has some of the functionality of MOV , CR0. See discussion there.

LMSW is also a legacy that, with a little luck, doesn't appear much in modern software (yeah, right).

There is no architectural protection for SMSW.

Resolution:

- raw – use scan & fix to replace with INT3 or other exception raising instruction
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, suffer the consequences (!)

2.15 LTR – Load Task Register

LTR uses a segment selector to load the Task Register from the GDT. The segment will be the Task State Segment (TSS) used as the destination for the task switch that LTR implements.

The whole TSS has to be protected by the supervisor, although it isn't clear, at least to me, to what extent it is really used by guest software of interest.

Execution of this instruction at CPL > 0 raises a #GP(0) exception.

Resolution:

- raw – emulate on interrupt
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, emulate on interrupt

2.16 STR – Store Task Register

STR returns the segment selector for the current TSS, as contained in the task register.

Since the supervisor will maintain the actual TSS, the results returned are likely to appear to be inconsistent with the setting the guest software thought it loaded into the task register. Some intervention must be made to return results that won't disturb the blissful ignorance of the guest.

There is no architectural protection for STR.

Resolution:

- raw – use scan & fix to replace with INT3 or other exception raising instruction
- cooked – translate/scan&fix to supervisor call or configure out
- demo – if it happens, suffer the consequences (!)

2.17 MOV CRx – Move to/from Control Register

2.17.1 CR0 – system control flags

The flags in CR0 control a number of features (paging, for instance) that may vary from guest to guest and from time to time within a given guest. There is also one state bit (TS) relating to the FPU, etc. The individual features are discussed in other sections of this paper.

Guest usage of CR0 clearly has to be fully hijacked by the supervisor.

2.17.2 CR2 – page fault linear address

See discussion of memory management issues below.

2.17.3 CR3 – Page-Directory Base (PDBR)

See discussion of memory management issues below.

2.17.4 CR4 – architectural extension flags

As with CR0, the bits in CR4 control a number of features (like PAE) whose usage may differ from guest to guest and time to time for a given guest. The individual features are discussed elsewhere.

These instructions, both directions, are available only when CPL = 0, generating a #GP(0) exception otherwise.

Resolution:

- raw – emulate on interrupt
- cooked – translate/scan&fix to supervisor call
- demo – if it happens, emulate on interrupt

2.18 MOV DRx – Move to/from Debug Registers

The debug registers control and support the debug and performance monitoring facilities. Supervisor issues are TBD – see separate discussion.

These instructions, both directions, are available only when CPL = 0, generating a #GP(0) exception otherwise.

Resolution:

- raw – emulate on interrupt
- cooked – translate/scan&fix to supervisor call
- demo – if it happens, emulate on interrupt

2.19 POPF/POPFD – Pop Stack into EFLAGS Register

The EFLAGS register contains three classes of flags.

The first is status information, a.k.a. the condition flags. This is harmless information which changes from instruction to instruction and is entirely the property of whatever code is running, including application code. The only concern for the supervisor is to ensure it's proper restoration when a virtual machine is restored.

The second class is also the property of the current code, but is control: the direction flag (DF). It has only to be saved and restored appropriately.

The third class contains system flags that control basic operation of the processor. Examples are the interrupt enable flag (IE) and I/O privilege level (IOPL). These must be under the control of the supervisor.

There is minimal architectural protection supplied by IA32. Some of the system flags are protected by various combinations of CPL and IOPL, but not all are.

Except perhaps in a "well done", perhaps "burnt", cooking mode, a shadow virtual EFLAGS register must be maintained and all attempts to write or read the register directly intercepted.

Resolution:

- raw – scan & fix
- cooked – translate/scan&fix to supervisor call
- demo – ?

2.20 PUSHF/PUSHFD – Push EFLAGS Register onto the stack

see discussion of POPF/POPFD above

2.21 CLI – Clear Interrupt Flag

The interrupt flag (IF) is represented by one of the bits in the EFLAGS register, and is a "system" flag. (See discussion under POPF/POPFD above).

It is protected by various combinations of state, principally CPL and IOPL.

2.22 STI – Set Interrupt Flag

see discussion of CLI above

2.23 VERR – Verify a Segment for Reading

VERR accepts as its source operand a segment selector and returns in the ZF flag whether the indicated segment is readable at the CPL in effect. There is no consultation of any page table information.

In most cases, this will be harmless. However, if the segment in question contains a system structure (page tables, LDT, etc.) being maintained as a shadow by the supervisor, there are potential problems. Given the unsophisticated use of segments in the guest software of interest, this instruction is probably the wrong place to attack any potential problems.

There is no architectural protection for VERR.

2.24 VERW – Verify a Segment for Writing

VERW accepts as its source operand a segment selector and returns in the ZF flag whether the indicated segment is writable at the CPL in effect. There is no consultation of any page table information.

See discussion of VERR.

There is no architectural protection for VERW.

2.25 LAR – Load Access Rights Byte

LAR accepts as its source operand a segment selector, and returns various access rights information from the segment descriptor.

Again, with limited use of segments, there may not be an issue here. See discussion at VERR. However, for instance, the DPL field becomes visible.

There is no architectural protection for LAR.

2.26 LSL – Load Segment Limit

LSL accepts as its source operand a segment selector, and returns the limit (bounds) information from the segment descriptor.

If all segments cover all visible memory, this is pretty uninteresting.

There is no architectural protection for LAR.

2.27 LDS/LES/LFS/LGS/LSS – Load Far Pointer

These instructions accept a far pointer (segment selector + offset), load the segment register indicated by the instruction and put the offset in the destination (register) operand location.

Since the segment descriptor must already be in the GDT or LDT, there is an issue here only if there are "hidden" entries in those tables.

Given the limited use of segments, are far pointers actually used?

The only architectural protection for these instructions is validation of CPL against RPL and DPL.

2.28 MOV Sreg – Load Segment Register

This instruction accepts a segment selector and uses it to load the indicated segment register.

The issues are the same as for LDS, et al.

2.29 MOV , Sreg – Store Segment Register

This instruction returns the segment selector from the indicated segment register.

This is a an issue only if the supervisor has placed hidden entries in the GDT and/or LDT and used them to spoof the guest. This is most likely with CS in a scan & fix situation, but given limited use of segments anyway, it may not really arise.

There is no architectural protection.

2.30 POP Sreg – Pop a Value from the Stack into a Segment Register

Same considerations as MOV into Sreg.

2.31 POPA/POPAD – Pop All General Purpose Registers

Same considerations as MOV into Sreg.

2.32 PUSH Sreg – Push Segment Register Onto the Stack

Same considerations as MOV from Sreg.

2.33 PUSHA/PUSHAD – Push All General Purpose Registers Onto the Stack

Same considerations as MOV from Sreg.

2.34 ENTER – Make Stack Frame for Procedure Parameters

Issues unknown.

2.35 LEAVE – High Level Procedure Exit

Issues unknown.

2.36 RDTSC – Read Time-Stamp Counter

TBD

2.37 MSRs – Model Specific Registers

TBD

3 Program Control Flow Issues

Most program flow instructions are only of interest to extent that they interact/interfere with scan & fix. In some case the far versions may be problematical.

3.1 Jcc – Jump If Condition Is Met

3.2 JMP – Jump

3.3 LOOP/LOOPcc – Loop According to ECX Counter

3.4 SYSENTER – Fast System Call

TDB

3.5 SYSEXIT – Fast Return From Fast System Call

TDB

3.6 CALL – Call Procedure

TBD

3.7 RET – Return From Procedure

TBD

4 Memory-Mapped I/O

TBD

5 Interrupts

TBD

5.1 INT – Call to Interrupt Procedure

TBD

5.2 BOUND – Check Array Index Against Bounds

TBD

5.3 IRET/IRETD – Interrupt Return

TBD

5.4 WAIT/FWAIT - Wait

TBD

6 Processor Mode Issues

TBD

7 Memory Management Issues

TBD

7.1 writing page tables

violates supervisor control and isolation
none
if B=C, just validate address, otherwise translate address

7.2 reading page tables

harmless if B=C, otherwise violates isolation
none
if B<>C, return translated info

8 Process Management Issues

TBD

8.1 idle loop

8.2 writing TSS

8.3 reading TSS

8.4 writing GDT

8.5 reading GDT

8.6 writing LDT

8.7 reading LDT

8.8 writing IDT

8.9 reading IDT

8.10 create/use task gate

9 FPU/MMX/SSE/SSE2 Issues

TBD

10 Debug & Performance Monitoring Issues

TBD

Title:	<Project Name>	23374200-004
Type:	Combined Functional and Detailed Design Document	Version B
Owner:	<Project Owner>	
Date:	12/20/00	Page 1 of 36

Docum nt Distribution List

	Name	Location		Name	Location
A	<name>	<address>	O	<name>	<address>
A	<name>	<address>	O	<name>	<address>
A	<name>	<address>	O	<name>	<address>
R	<name>	<address>	O	<name>	<address>
R	<name>	<address>	O	<name>	<address>
R	<name>	<address>	O	Product Information Document Library	MV 226

Legend:		
A	-	Approver
R	-	Required Reviewer
O	-	Optional Reviewer

________ Approved Without Change

________ Approved With Indicated Changes

________ Not Approved

____

Signature of Approver / Date

________ Comments

________ No Comments

____

Signature of Reviewer / Date

Comments due by: 01/04/01

Notice : The printed version of this document may not be current.
Verify the version date against the on-line file in the Design Review System DataBase.

Abstract: This pap r describes the design and implementation of the Windows CE virtual disk driver on top of Extensible Firmware Interface for quick demo of the Supervisor concepts.

UNISYS

Supervisor Disk Driver

Combined Functional and Detailed Design Document

Date: 12/20/00

23374200-004
Version B

Approval Status: Being Written

Name	Role	Location	Mail Stop	Phone
Kaike Wan	Author	Malvern	B233	(610) 648-2124
John Landis	Team Leader	Malvern	A29F	(610) 648-2497
Alan Grubb	Team Leader	Malvern	B249	(610) 648-4103
Terry Powderly	Consultant	Malvern	A29M	(949) 380-4062

This page contains hidden text, which includes the Document Info Table and the macros used to update the table, restore bookmarks in the table, and remove hidden blue text when the document is complete. To view hidden text in Microsoft Windows, turn on the option to view hidden text.

Table of Contents:

1. Docum nt Control

This document was generated using the PPG Template Generator, 3490 3880, revision H. This document is based on the combination of the PPG Functional Design Specification Guide and Template (3486 2417) and the PPG Detailed Design Specification Guide and Template (3486 2433).

NOTE: This document may be reviewed in stages as information is added. For example, one version of this document may contain only requirement information. A later version will also have proposal information. Therefore, earlier versions of the document may contain blank sections if these sections do not pertain to design phase that is being documented. Please refer to the PPG documents listed above to see what information/sections are pertinent to a particular design phase.

1.1. Change History

Based on the presentation prepared by Terry Powderly, the Supervisor Task List document written by John Landis, and discussion with Raj Subrahmanian. The document can be inserted into *Section 2.4 vDisk, Supervisor Task List.*

1.2. Document Cross Reference

This document takes inputs from various sources, some of which are listed below:

Title	Document-ID
Supervisor Task List	None.
Supervisor Powerpoint Presentation by Terry Powderly	None
MSDN	None

2. Introduction

2.1. Purpose

2.2. Scope

2.3. Key Points

Here are some extracts from Microsoft Web site as well as from MSDN:

Windows Embedded consists of three products: Windows CE .NET is the modular real-time embedded operating system for small footprint and mobile 32-bit intelligent and connected devices. Windows XP Embedded allows embedded system developers to reduce their time to market and easily create mid-range to high-end devices that integrate with an existing IT infrastructure. Windows 2000 with Server Appliance Kit enables you to build a wide variety of server appliances, including Networked Attached Storage (NAS) appliances, Web server appliances, and Windows-based Terminals (WBT).

Windows CE implements device drivers as dynamic-link libraries (DLLs).

Device Manager (device.exe) loads boot time device drivers at system initialization by enumerating branches under a special key in the system registry, calls their initialization routine, and allocates resources.

Device drivers can also be loaded after system initialization, through explicit calls to interfaces exposed by Device Manager.

Windows CE drivers are two-layer architectures that isolate hardware-specific requirements from interface requirements. The functionality of the driver is exposed to the operating system and applications through the Model Device Driver (MDD), which exposes a Device Driver Interface (DDI) suited to the type of device. The Platform Dependent Driver (PDD) layer is written directly to the hardware and exposes a Device Driver Service Provider Interface (DDSI) that is prescribed by the MDD.

If you want your driver to interface with the file allocation table (FAT) file system or file system driver (FSD) Manager, your block driver should expose the stream interface.

Device drivers for block devices generally expose the stream interface, and the block devices appear as ordinary disk drives.

The FAT file system accesses the block device by calling the block device driver's XXX IOControl function with the appropriate I/O control codes.

You should include registry keys under the platform's **HKEY_LOCAL_MACHINE\Drivers\Builtin** registry key so that the Device Manager loads the block device driver when the platform starts.

3. Interdependencies

This implementation is only one components of the Unisys Supervisor architecture for feasibility demonstrations, and it depends on the successful configuration of a working Windows CE 3.0 as a loadable guest OS on the ES7000 system.

4. Functional Overview

4.1. Major Functions

For Windows CE, the Supervisor Virtual Disk Driver will be largely based on Microsoft's implementation of RAM disk driver (Ramdisk.dll) to keep hand off physical disk for quick development. As the Supervisor Virtual LAN implementation is sufficient to provide networking capability for each guest OS, the vDisk driver can take advantage of this and connects to a remote storage service in a way similar to Unisys Console Manager 2 to provide virtual disk, floppy, and CD-ROM services to the Guest OS. This latter part will be more likely implemented in another driver targeting desktop or server Windows OS instead of the embedded versions that are only intended for feasibility demonstration.

The mechanism of the Supervisor vDisk driver is implemented in four components: the stream interface, the internal disk I/O functions, the registry interface, and the interface to the disk manager. The stream interface is a required interface between the operating system--mainly the file system, and this driver to fulfill the requirements for the two-layer driver model wherein the file system driver implements the Model Device Driver (MDD) that exposes the Device Driver Interface (DDI) to applications and other modules in the system while this driver implements the Platform Device Driver(PDD) that exposes the Device Driver Service Provider Interface (DDSI) to the MDD. The required stream interface functions included DSK_Init(), DSK_Deinit(), DSK_Open(), DSK_Close(), DSK_PowerUp(), DSK_PowerDown(), DSK_IOControl(), which will be described in details later.

The registry interface will have functions to set registry keys and values for this driver to be loaded by the Device Manager at system initialization and also functions to retrieve parameters from the registry for the driver to work properly.

The internal disk I/O functions will implement the mechanism that the stream interface functions depend on. It will have data structures to track disk state, disk parameters, open file handles to the disk, etc. It will have functions to get/set the disk parameters. More importantly, it implements the function for disk I/O operations that DSK_IOControl() needs. This component of the vDisk driver invokes the functions in the registry interface to retrieve operation parameters from the registry during driver initialization.

The Disk Manager controls and manages the disk accesses for all Guest OSes. As described above, the DM will provide functions that the Guest OSes should use through a stub for all disk accesses as well as a cache memory block for each Guest OS. Internally, the DM should synchronize the disk accesses from all Guest OSes by inserting the requests into a queue or multiple queues of various priorities. A worker thread then consumes the request queue(s) and fulfills the requests one by one. It is desirable that the DM has a larger cache memory and also provides the capability to combine some of the requests. For example, if Guest OS1 requests to read 10 sectors from Logical Block Address (LBA) 325 and Guest OS2 tries to read 32 sectors from LBA 330, then the DM can satisfy both requests by reading 37 sectors starting from LBA 325 in one disk access, which is usually the performance bottleneck. It is even better that we can provide prefetch caching mechanism, i.e., to read more than a Guest OS requests. In the example above, the DM may read 64 sectors from LBA 325, which will save another disk access if Guest OS1 requests to read another 10 sectors from LBA 335. Through the DM, it is also easy to implement asynchronous I/O. The vDisk driver in the Guest OS passes a disk access request along with an event to the DM and returns to do its own work without waiting for the disk access to complete, but keeps checking the event status from time to time. Upon the completion of the disk access, the DM signals the event. After a while, the vDisk driver comes around to find that the disk I/O is done and can then signal the completion to its upper layer driver or application.

To access the disk, the DM has a number of choices. The first choice is that the DM makes use of the Supervisor EFI runtime services that contains disk (IDE or SCSI) drivers implementing EFI DISK_IO and BLOCK_IO protocols. These drivers should be based on the Intel sample code for boot time services with minor changes. The key is that the DM must serialize all accesses to the same physical disk to avoid any conflict. Another solution is the Unisys Storage Sentinel, where a SAN manager provides virtual disks to the Supervisor Disk Manager. In this case, the DM can delegate all or at least part of the disk access management task to the SAN Manager. More details about SAN Manager will be needed for further meaningful discussion. Another solution is to leverage the Supervisor Virtual LAN capability and the Unisys Console Manager 2 remote storage service. In this scenario, the DM connects to a remote storage service through network and sends its disk access requests to the storage service, where the accesses to a physical disk take place.

For the prototype, the DM will provide no access to a physical disk. Instead a memory block will be reserved. For any disk write from Guest OSes, the data will be copied from Guest OS's local cache memory to this cache.

4.2. Assumptions

It is assumed that the demo will use Windows CE 3.0 as the Guest OS and only a virtual disk is needed, which requires no accesses to a physical disk. The vDisk driver can be bundled into the OS by the Windows CE Platform Builder.

5. Functional Description

5.1. vDisk Driver Functions

5.1.1. vDisk Stream Interface

5.1.1.1. DSK_Init

This function is called by the Windows CE Device Manager to initialize a stream interface device.

Prototype:

```
DWORD DSK_Init(
     DWORD dwContext
    );
```

Note: for Windows CE .Net, the prototype is

```
DWORD DSK_Init(
    LPCTSTR pContext,
    LPCVOID lpvBusContext
    );
```

Parameters:

dwContext (or pContext) -- a pointer to the register path to the active key for the stream interface driver.

lpvBusContext -- The potentially process-mapped pointer passed as the fourth parameter to ActivateDeviceEx. If this driver was loaded through legacy mechanisms, then *lpvBusContext* is zero. This pointer, if used, has only been remapped as it passes through the protected server library (PSL). Also, any pointers referenced through *lpvBusContext* must be remapped with MapCallerPtr before they can be dereferenced.

Return:

0 – The function call failed.

Other – A handle to the device context created, which will be passed to other stream interface functions such as DSK_Open, DSK_DeInit, DSK_PowerUp, DSK_PowerDown.

Description:

The Device Manager calls this function to initialize the virtual disk as the result of the call to ActivateDeviceEx when the user starts to use this virtual disk. The Device Manager specifies a pointer to a string containing the registry path to the active key of the virtual disk device in the *pContext* parameter. Usually, the string contains the registry path to a numbered subkey of the **HKEY_LOCAL_MACHINE\Drivers\Active** key. This function allocates memory to store the disk information and initializes the disk information data structure by setting the initial disk state and copying the registry path to the disk information data structure. Through a TBS mechanism, this function also maps the physical memory for the disk cache into its process space and keeps a pointer to it in the disk information data structure. Upon exit, the pointer to the allocated disk information data structure is returned.

5.1.1.2. DSK_Deinit

Device Manager invokes this function as a result of calling DeregisterDevice when the user stops using the vDisk.

Prototype:

```
    BOOL DSK_Deinit(
        DWORD dwContext
    );
```

Parameter:

dwContext – Handle to the device context returned from the call to DSK_Init, a pointer to the disk information data structure in this case.

Return:

True indicates success. False indicates failure.

Description:

This function calls DSK_Close to reduce the open handle count, and then frees the cache buffer and the disk information data structure.

5.1.1.3. DSK_Open

This function is indirectly invoked when the application opens the vDisk for read or write by calling CreateFile.

Prototype:

```
    DWORD DSK_Open(
         DWORD dwData,
        DWORD dwAccess,
        DWORD dwShareMode
    );
```

Parameters:

dwData – Handle to the device context returned from DSK_Init, a pointer to the disk information data structure in this case.

dwAccess -- Specifies the requested access code of the device. The access is a combination of read and write.

dwShareMode -- Specifies the requested file share mode of the vDisk device. The share mode is a combination of file read and write sharing.

Return:

This function returns the open context of the device that can be used in calls to DSK_Read, DSK_Write, DSK_Seek, and DSK_IOControl. If the device cannot be opened, NULL is returned.

Description:

This function first checks if dwData is a valid pointer to the disk information data structure, and then continues initializing the disk information data structure. After increasing the open count of the disk and changing its state to open state, this function returns a pointer to the disk information data structure. If for any reason any of the checks fails, NULL is returned.

5.1.1.4. DSK_Close

This function is called by the operating system when the application invokes CloseHandle to close the handle to this stream interface device.

Prototype:

```
BOOL DSK_Close(
    DWORD Handle
);
```

Parameter:

Handle – The handle returned from DSK_Open, used to identify the open context of the device.

Return:

False indicates failure while true indicates success.

Description:

This functions first checks if the passed parameter is a valid disk pointer. If the parameter is a valid disk pointer, it checks if its current state is open and reduces its open count by one. If the resulting open count equals 0, it returns true. For all other cases, it returns false.

5.1.1.5. DSK_PowerUp

The operating system calls this function to restore power to the device.

Prototype:

```
void DSK_PowerUp(
    DWORD hDeviceContext
);
```

Parameter:

hDeviceContext – The device context returned from DSK_Init.

Return:

None.

Description:

vDisk has an empty implementation for this function since no physical device is directly managed by this driver.

5.1.1.6. DSK_PowerDown

This function is invoked by the operating system to suspend the power to the device.

Prototype:

```
void DSK_PowerDown(
    DWORD hDeviceContext
);
```

Parameter:

hDeviceContext – The device context returned from DSK_Init.

Return:

None.

Description:

This function sets the state of the disk to be dead so that it can not be accessed until it is powered up again.

5.1.1.7. DSK_IOControl

This function is responsible for the file system requests to read and write data, to get and set media information, to get device names, and to perform low level formatting. For Block device driver, the functions DSK_Read, DSK_Write, and DSK_Seek are not used by the file system to accommodate an application's read/write requests.

Prototype:

```
BOOL DSK_IOControl(
     DWORD Handle,
    DWORD dwIoControlCode,
    PBYTE pInBuf,
     DWORD nInBufSize,
    PBYTE pOutBuf,
    DWORD nOutBufSize,
    PDWORD pBytesReturned
 );
```

Parameters:

Handle – The handle returned from DSK_Open, used to identify the open context of the device.

dwIoControlCode – The I/O control code for the operation to perform.

pInBuf -- Pointer to the buffer containing data to be transferred to the device.

nInBufSize -- Specifies the number of bytes of data in the buffer specified for *pBufIn*.

pOutBuf -- Pointer to the buffer used to transfer the output data from the device.

nOutBufSize -- Specifies the maximum number of bytes in the buffer specified by *pBufOut*.

pBytesReturned – Pointer to the DWORD buffer that this function uses to return the actual number of bytes received from the device.

Return:

TRUE indicates success. FALSE indicates failure.

Description:

A block device driver must support the following I/O Control code: IOCTL_DISK_FORMAT_MEDIA, IOCTL_DISK_FORMAT_VOLUME, IOCTL_DISK_GET_STORAGEID, IOCTL_DISK_GETINFO, IOCTL_DISK_GETNAME, IOCTL_DISK_INITIALIZED, IOCTL_DISK_READ, IOCTL_DISK_SCAN_VOLUME, IOCTL_DISK_SETINFO, IOCTL_DISK_WRITE. Upon entry, this function checks the validity of the passed-in parameters including the open context. Depending on the I/O control code, it dispatches the request to its corresponding function inside the vDisk Internal function.

5.1.2. vDisk Registry Interface

5.1.2.1. OpenDriverKey

This function open the registry key for the vDisk driver so that it can be used further to get other configuration parameters for the vDisk driver.

Prototype:

```
HKEY OpenDriverKey(

    LPTSTR ActiveKey

);
```

Parameter:

ActiveKey – The path to the active subkey created by the Device Manager under "***HKEY_LOCAL_MACHINE\Drivers\Active***", which is passed to the DSK_Init function when the Device Manager loads this driver.

Return:

This function returns the key to vDisk driver registry entry under "***HKEY_LOCAL_MACHINE\Drivers\BuiltIn***" upon success. Otherwise, it returns NULL.

Description:

The registry settings for the vDisk driver is created under "***HKEY_LOCAL_MACHINE\Drivers\BuiltIn\vDisk***". At system start, the Device Manager calls the Registry Enumerator to enumerate all drivers under "***HKEY_LOCAL_MACHINE\Drivers\BuiltIn***". For each entry under this key, the Enumerator loads the DLL indicated by the entry's DLL value, and then creates a subkey for the driver under "***HKEY_LOCAL_MACHINE\Drivers\Active***". Then it calls the Init function or the driver's entry point and passes in a string containing that subkey under "***HKEY_LOCAL_MACHINE\Drivers\Active***". This subkey contains the string corresponding to the registry key that Enumerator originally encountered under "***HKEY_LOCAL_MACHINE\Drivers\BuiltIn***", i.e., "***HKEY_LOCAL_MACHINE\Drivers\BuiltIn\vDisk***" in this case. This function calls RegOpenKeyEx to open this active subkey, retrieves the key value by RegQueryValueEx, uses the obtained registry path to open the key to its driver registry entry under "***HKEY_LOCAL_MACHINE\Drivers***", and returns the key. A NULL will be returned if any error occurred.

5.1.2.2. G tDiskSize

This device driver calls this function to obtain the size of the disk.

Prototype:

```
DWORD GetDiskSize(
    PDISK pDisk
);
```

Parameter:

pDisk – A pointer to the disk information data structure.

Return:

The size of the disk set under the registry entry "*HKEY_LOCAL_MACHINE\Drivers\BuiltIn\vDisk*" by the value "Size" if successful; the default value (1 MB) otherwise.

Description:

This function calls OpenDriverKey to open its driver registry entry with the active key path kept in its disk information data structure, and queries the given size of the disk it is going to manage.

5.1.3. vDisk Internal Functions

5.1.3.1. GetDeviceInfo

The DSK_IOControl function invokes this function as a result of an application's calling DeviceIoControl with the code IOCTL_DISK_DEVICE_INFO.

Prototype:

```
BOOL GetDeviceInfo(
    PDISK pDisk,
    PSTORAGEDEVICEINFO pInfo
);
```

Parameters:

pDisk – A pointer to the disk information data structure.

pInfo – A pointer to the STORAGEDEVICEINFO structure that contains the information of the disk device, such as device class, device type, device access flag (read/write), and profile name.

Return:

True for success; false for failure.

Description:

If pDisk and pInfo are not NULL, this function opens the its drive registry key by calling OpenDriverKey and queries for the "Profile" value. If the query fails (the value may not exist), it copies the default string "Default" to the profile name of pInfo. After setting the device class, device type, and read/write flag, it returns True. If either pDisk or pInfo is NULL or any error occurs during the registry query, it returns False.

5.1.3.2. GetFolderName

The FAT file system uses IOCTL_DSK_GETNAME to request for the name of the folder that determines how end users (applications) access the block device. Consequently DSK_IOControl dispatches the request to this function.

Prototype:

```
    BOOL GetFolderName(
        PDISK pDisk,
         LPWSTR FolderName,
        DWORD cBytes,
        DWORD * pcBytes
    );
```

Parameters:

pDisk -- A pointer to the disk information data structure.

FolderName – The buffer for the driver to fill with the folder name in Unicode.

cBytes – The size of FolderName in bytes.

pcBytes – The address of a DWORD that will receives the length of the returned string plus the terminating NULL character in bytes.

Return:

True if success; False otherwise.

Description:

This function opens the driver registry entry by OpenDriverKey, and then queries for the "Folder" value. If the operation fails, the return length (*pcBytes) will be set to 0.

5.1.3.3. CloseDisk

This is a utility function that frees all resources associated with a disk.

Prototype:

```
    VOID CloseDisk(
        PDISK pDisk
    );
```

Parameter:

pDisk -- A pointer to the disk information data structure.

Return:

None.

Description:

This function removes pDisk from the global disk list, deletes the critical section inside the disk information data structure, frees the allocated cache buffer, and finally frees the disk information data structure.

5.1.3.4. SetDiskInfo

A Block device driver responds to IOCTL_DISK_SETINFO control code to service FAT file system requests to set disk information.

Prototype:

```
    DWORD SetDiskInfo(
        PDISK pDisk,
        PDISK_INFO pInfo
    );
```

Parameters:

pDisk – A pointer to the disk information data structure.

pInfo – The buffer contains the DISK_INFO structure to set.

Return:

ERROR_SUCCESS.

Description:

The FAT file system obtains the disk info by calling DeviceIoControl with IOCTL_DISK_GETINFO. When it finds a discrepancies between this data and those obtained from the medium, it sets the disk info through DeviceIoControl with code IOCTL_DISK_SETINFO, which is dispatched to this function. This function copies the new disk info into the disk information data structure and returns.

5.1.3.5. GetDiskInfo

The file system invokes this function to get the disk information through DeviceIoControl with IOCTL_DISK_GETINFO (or DISK_IOCTL_GETINFO in Windows CE 3.0).

Prototype:

```
    DWORD GetDiskInfo(
        PDISK pDisk,
        PDISK_INFO pInfo
    );
```

Parameters:

pDisk – A pointer to the disk information data structure.

pInfo – A pointer to a DISK_INFO structure to receive the disk information.

Return:

ERROR_SUCCESS.

Description:

This function copies the disk information inside the disk information data structure to pInfo. The parameter checking has been done in the DSK_IOControl, which dispatches the request to this function.

5.1.3.6. DoDiskIO

This function handles the I/O Control code DISK_IOCTL_READ and DISK_IOCTL_WRITE dispatched from DSK_IOControl.

Prototype:

```
DWORD DoDiskIO(
    PDISK pDisk,
        DWORD Opcode,
        PSG_REQ pSgr
    );
```

Parameters:

pDisk -- A pointer to the disk information data structure.

Opcode – I/O Control code (DISK_IOCTL_READ or DISK_IOCTL_WRITE).

pSgr – A pointer to the SG_REQ structure that contains the scatter/gather buffers.

Return:

Standard disk request status. If the request is processed successfully, ERROR_SUCCESS is returned. Some other error codes include ERROR_INVALID_PARAMETER and ERROR_SECTOR_NOT_FOUND.

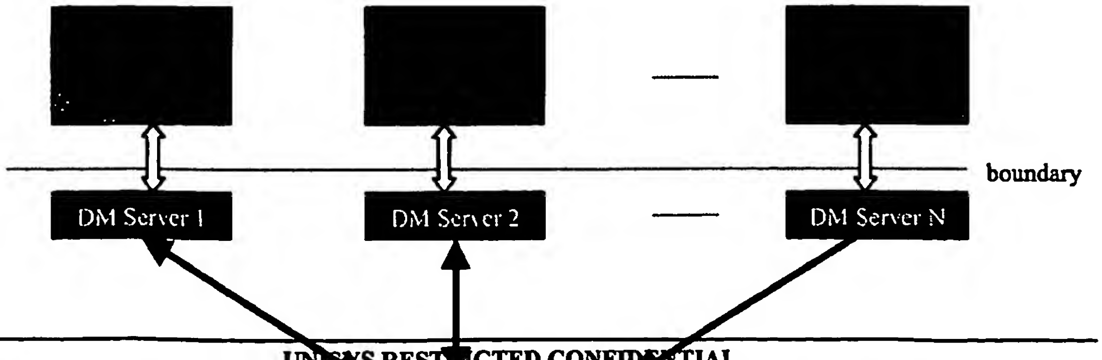
Description:

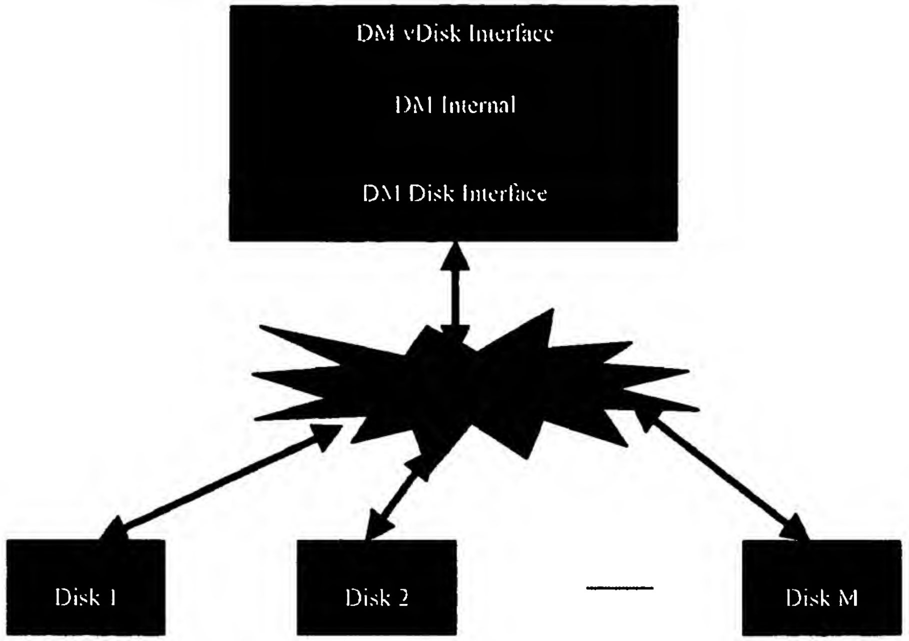
This function is invoked when DSK_IOControl receives DSK_IOCTL_READ or DSK_IOCTL_WRITE. It checks the incoming parameters to make sure that the number of scatter/gather buffers does not exceed the predefined maximum limit and the requested sectors are within the disk range. For read request, it will submit the request to the Disk Manager. Once the data is back, it copies the data into the scatter/gather buffers based on individual buffer size. For write request, it combines the write data from the scatter/gather buffers into its own cache buffer and submits one request to the Disk Manager.

5.2. Disk Manager Functions

5.2.1. Architecture

The storage architecture for the Supervisor project is shown in the diagram below. In this diagram, vDisk N is the vDisk driver for Guest OS N. DM Stub is vDisk driver's interface to the Disk Manager functions. Together with a DM Server, the DM Stub facilitates function calls across the memory address boundary, in a way similar to the data marshaling in Microsoft's COM. This is required because Disk Manager functions are running in either the Supervisor Core Firmware environment or a separate virtual machine environment and are not in the same virtual memory space with any vDisk driver in a Guest OS environment. Consequently, the passing of parameters between the vDisk driver and the Disk Manager must be in physical memory addresses and must follow a custom calling convention established by the DM Stub – DM Server interface. DM vDisk Interface exposes the Disk Manager functions that are directly called by the DM Servers. These functions will insert disk access requests into queues processed by the Disk Manager Internals (DM Internals), which will in turn call the DM Disk Interface to access physical disks. The DM Disk Interface will shield DM Internal from the details of the underlying disk system.





There should be a linked list of DM service requests in the Guest OS and the list head is known to both vDisk driver in this Guest OS and the Disk Manager. Whenever the vDisk driver receives a disk access request, it queues a request packet into the linked list and raise a request event. A DM worker thread, one for each Guest OS, waits for this event to be signaled. Once awakened, the DM worker thread walks through the linked list to get each request, parses its parameters, and calls the appropriate functions. After the function returns, the DM worker thread sets the status code so that vDisk driver can complete this request.

5.2.2. vDisk – DM Stub Interface

The DM Stub exposes most of the service functions of the Disk Manager vDisk Interface for the vDisk driver to access disk, namely, dmAllocateCache, dmFreeCache, dmReadBlocks, dmWriteBlocks, dmIOControl. For details about each service function, see the DM vDisk Interface. For any disk I/O, the vDisk driver should call these functions directly as if these functions were implemented within the Guest OS environment, without any concern about the parameter marshalling.

5.2.3. DM Stub – DM Server Interface

This is where all parameter marshalling occurs. In this interface, for each function call, the DM Stub sets up the parameters in a request, queues the request to a request queue, gets back the I/O status and any associated data, and returns the result to the vDisk driver. On the other hand, the DM Server processes the request queue, retrieves the parameters for each request, calls the exported service functions in DM vDisk Interface to serve the request, and returns the result to the DM Stub. During this process, two of the key issues are how to pass parameters and result back and forth, and how to synchronize the work of the DM Stub and the DM Server.

5.2.3.1. Parameter Marshalling

The requests are placed in the queue vDiskRequestQ before processing and put into the queue vDiskDoneQ after processing. There must be two queue heads in the shared memory area between the Guest OS and the Supervisor Core Firmware environment that both Disk Manager and the vDisk must be able to access. The format of the entire shared memory area is TBS. The defines are as follows:

```
UINT32          vDiskRequestQHead;
UINT32          vDiskDoneQHead;
```

The queue head vDiskRequestQHead points to the physical memory address of the next request packet while vDiskDoneQHead points to the physical memory address of the next done request packet. If any of the queue is empty (no request or no request is done yet), the corresponding queue head should be 0.

The request packet is define here:

```
typedef struct {
    UINT32          signature;      // From byte 1 to byte 3: 'S', 'U', 'P', 'D'.
    UINT32          size;           // Size of the total packet in bytes, including any function
                                    // specific parameters.
    UINT32          nextRequest;    // The physical address of next request
    DM_FUNC_CODE    funcCode;       // Function code(UINT8): which function to call
    UINT8           bCancelled;     // If this request has been cancelled by the caller
    UINT16          statusCode;     // Status code: NOT ACCESSED on input.
    UINT32          parameters;     // Physical address of the function specific parameters
    } VDISK_REQUEST;
```

The request packet is followed by any function specific parameters. It should be reminded that all pointers must use physical memory address instead of virtual address. A stub functions allocates a request packet from a vDisk driver memory pool, initializes it with the calling parameters, inserts it into the linked list vDiskRequestQ, and waits for its to come back in the vDiskDoneQ. The vDisk driver has an internal structure to map each request packet to its upper-layer request. Either synchronous or asynchronous strategy could be implemented in the stub functions. The DM Server processes the vDiskRequestQ, parses the parameters for each request, and then calls the appropriate DM service function based on the requested function code and parameters. Upon the completion of the request (the return of the DM service function), the DM Server puts the finished request into the vDiskDoneQ for the DM Stub to complete the cross-boundary call.

Related definition:

```
typedef     UINT8           DM_FUNC_CODE;
```

5.2.3.2. Synchronization

Three control flags are needed in the Shared Memory area described above to synchronize the processing of DM Stub and DM Server, as defined below:

```
UINT8           vDiskRequestQState;         // 0:  Request Queue not ready for processing yet.
```

```
                                              // 1: Request Queue ready for processing
      UINT8          vDiskDoneQState;         // 0: DM Server can add Done request to it.
                                              // 1: DM Server can not touch the done queue.
```

5.2.4. DM vDisk Interface

5.2.4.1. dmGetDMServices

This function returns a pointer to the function table exposed by the Disk Manager to the Guest OSes.

Prototype:

```
    DM_STATUS dmGetDMServices(
        OUT DM_HANDLE          *hDisk,
        OUT SM_SERVICES        **dmServices
        );
```

Parameters:

hDisk – The address of a variable that will receive the handle to the Disk Manager. The handle must be used in all subsequent calls to other DM service functions.

dmServices -- The address of a pointer that will points to the Disk Manager function table.

Related definitions:

```
    #define   OUT
    #define    IN
    typedef   UINT32          DM_HANDLE;
    typedef   struct {
    DM_GETDMSERVICES dmGetDMService;
    } DM_SERVICE;
```

Status code:

DM_SUCCESS -- The physical address of the DM function table is successfully returned.

DM_SVC_NOT_AVAILABLE – The DM services are not available.

DM_INVALID_PARAMETER –At least one of the parameters is invalid.

Description:

This function could be bundled into Supervisor Runtime Services and be given to the Guest OS, which will in turn hand this function to the vDisk driver to retrieve the physical address of the Disk Manager function table from it. The calling vDisk driver must then map the DM function table into its virtual address space to be able to call the service functions. If the status code is DM_SUCCESS, dmServices will contain the valid physical address of the DM function table while hDisk contains the Disk Manager handle; otherwise, they are invalid and can not be used later. For all subsequent accesses to the DM functions, the vDisk driver must present this DM handle to identify itself properly. The Disk Manager will use the handle to index its internal context (status) information for all the vDisk drivers that rely on its services.

5.2.4.2. dmAllocateCache

The vDisk driver in the Guest OS calls this function to get a range of physical memory for caching.

Prototype:

```
DM_STATUS dmAllocateCache(
    IN DM_HANDLE                    hDisk,
    IN DM_CACHE_ALLOCATE_TYPE       type,
    IN OUT DM_PHYSICAL_ADDRESS      *startAddress,
    IN OUT DM_SIZE                   *pages
    );
```

Parameters:

hDisk – Handle for this vDisk driver, which is returned in dmGetDMServices().

type – Cache allocation type.

startAddress – The address of a variable which will receive the starting physical address of the assigned memory range.

pages – The address of a variable that will receive the size of the assigned memory range in pages.

Related definitions:

```
typedef     enum {
    CACHE_ALLOCATE_DEFAULT,
    CACHE_ALLOCATE_ADDRESS,
    CACHE_ALLOCATE_ANY
    } DM_CACHE_ALLOCATE_TYPE;
typedef     UINT32          DM_SIZE;
```

Status code:

DM_SUCCESS – The cache is successfully allocated.

DM_INVALID_PARAMETER – One or more of the parameters (type, startAddress, size) are not valid.

DM_INVALID_HANDLE – The handle hDisk is invalid.

DM_OUT_OF_RESOURCES – The Disk Manager is out of resources (memory).

DM_NOT_FOUND – The requested start address could not be found.

Description:

This function allocates a block of memory for caching for the calling vDisk driver with a minimum size of one page (4K). Depending on the cache allocation type, the caller may have to provide some valid parameters on input. For CACHE_ALLOCATE_DEFAULT, both startAddress and pages will be ignored on input. The DM will allocate a block of memory of a default size at any available address. On output, startAddress will point to the physical address of the allocated pages and the pages point to the number of pages allocated. The default size is determined by the Disk Manager, based on system configuration. If the allocation type is CACHE_ALLOCATE_ADDRESS, startAddress must point to a valid physical address and pages must point to a valid value. Otherwise, DM_INVALID_PARAMETER or DM_NOT_FOUND will be returned. Allocation requests of type CACHE_ALLOCATE_ANY will allocate any available pages of the requested size at any available physical address. The parameter startAddress is ignored on input in this case, but will point to the allocated physical address on output. In any case, neither startAddress nor pages can be NULL.

5.2.4.3. dmFreeCache

The Guest OS should call this function to free the cache memory allocated in a previous call to dmAllocateCache().

Prototype:

```
DM_STATUS dmFreeCache(
    IN  DM_HANDLE                hDisk,
    IN DM_PHYSICAL_ADDRESS       startAddress,
    IN  DM_SIZE                  pages
);
```

Parameters:

hDisk – Handle for this vDisk driver, which is returned in dmGetDMServices().

startAddress – The start physical address of the cache to be freed.

pages – The number of pages to free. This number and startAddress must match those allocated in a previous call to dmAllocateCache.

Status codes:

DM_SUCCESS – The cache is successfully freed.

DM_INVALID_PARAMETER – One or both of the parameters (startAddress, pages) are not valid.

DM_INVALID_HANDLE – The handle hDisk is invalid.

DM_NOT_FOUND – The cache is not allocated by dmAllocateCache.

Description:

This function can only free a cache (memory) allocated through a previous call to dmAllocateCache, which demands that the parameters startAddress and pages must be the same as those values obtained from dmAllocateCache.

5.2.4.4. dmReadBlocks

This function is the primary channel for disk read accesses, wherein the caller requests for a certain number of blocks (sectors) from a logical block address (LBA).

Prototype:

```
DM_STATUS dmReadBlocks(
    DM_HANDLE            hDisk,
    DM_LBA               lba,
    DM_SIZE              blocks,
    UINT8                *buffer,
    );
```

Parameters:

hDisk – Handle for this vDisk driver, which is returned in dmGetDMServices().

lba – The start logical block address of the blocks (sectors) to read.

blocks – The number of blocks to read.

buffer – The buffer to receive the read data. It must be large enough to contain the requested data.

Status codes:

DM_SUCCESS – The blocks are successfully read.

DM_INVALID_PARAMETER – One or more of the parameters (lba, blocks, buffer) are not valid.

DM_INVALID_HANDLE – The handle hDisk is invalid.

DM_DEVICE_ERROR – The device reported an error while attempting to perform the read operation.

Description:

This function returns the number of blocks (sectors) requested by the caller. If not all the data are read, an error is returned. Sanity check is performed on the given parameters.

5.2.4.5. dmWriteBlocks

This function writes the data to designated logical block address.

Prototype:

```
DM_STATUS dmWriteBlocks(
    DM_HANDLE           hDisk,
    DM_LBA              lba,
    DM_SIZE             blocks,
    UINT8               *buffer,
    );
```

Parameters:

hDisk – Handle for this vDisk driver, which is returned in dmGetDMServices().

lba – The start logical block address of the blocks (sectors) to write.

blocks – The number of blocks to write.

buffer – The buffer containing the data to write. It must be large enough to contain the requested data to write.

Status codes:

DM_SUCCESS – The blocks are successfully written.

DM_INVALID_PARAMETER – One or more of the parameters (lba, blocks, buffer) are not valid.

DM_INVALID_HANDLE – The handle hDisk is invalid.

DM_DEVICE_ERROR – The device reported an error while attempting to perform the write operation.

DM_WRITE_PROTECTED – The device can not be written to.

Description:

This function writes the number of blocks (sectors) requested by the caller. If not all the data are written, an error is returned. Sanity check is performed on the given parameters.

5.2.4.6. dmFlushBlocks

This function flushes all modified blocks to a physical disk.

Prototype:

```
DM_STATUS dmFlushBlocks(
    DM_HANDLE              hDisk,
    );
```

Parameters:

hDisk – Handle for this vDisk driver, which is returned in dmGetDMServices().

Status codes:

DM_SUCCESS – all modified blocks are successfully written to the physical disk.

DM_INVALID_HANDLE – The handle hDisk is invalid.

DM_DEVICE_ERROR – The device reported an error while attempting to perform the write operation.

Description:

All data modified prior to the flush must be written the physical disk device before DM_SUCCESS is returned. This means that all modified blocks in all caches must be flushed.

5.2.4.7. dmReset

This function resets the hardware disk.

Prototype:

```
DM_STATUS dmReset(
    DM_HANDLE              hDisk,
    BOOLEAN                extendedVerification
    );
```

Parameters:

hDisk – Handle for this vDisk driver, which is returned in dmGetDMServices().

extendedVerification – Indicates that Disk Manager may perform a more exhaustive verification operation of the hardware disk during reset.

Status codes:

DM_SUCCESS – The hardware disk is successfully reset.

DM_INVALID_HANDLE – The handle hDisk is invalid.

DM_DEVICE_ERROR – The device is not functioning properly and can not be reset.

Description:

The Disk Manager must take whatever necessary steps to reset the hardware disk.

5.2.4.8. dmIOControl

This function is responsible for device I/O control operations such as IOCTL_GET_DISK_GEOMETRY that queries the attributes of the disk.

Prototype:

```
DM_STATUS dmWriteBlocks(
    DM_HANDLE               hDisk,
    DM_IOCTL_CODE           code,
    VOID *                  inBuffer,
    DM_SIZE                 inSize,
    VOID *                  outBuffer,
    DM_SIZE                 outsize,
    DM_SIZE *               bytesReturned
    );
```

Parameters:

hDisk – Handle for this vDisk driver, which is returned in dmGetDMServices().

code – The IOCTL code.

inBuffer – The input data buffer.

inSize – The size of the input data in inBuffer.

outBuffer – The output data buffer.

outSize – The maximum size of output data that can written to outBuffer.

BytesReturned – Number of bytes returned.

Status codes:

DM_SUCCESS – The IOCTL operation is successfully performed.

DM_INVALID_HANDLE – The handle hDisk is invalid.

DM_DEVICE_ERROR – The device reported an error while attempting to perform the IOCTL operation.

DM_INVALID_PARAMETER – One or more of the parameters (code, inBuffer, inSize, outBuffer, outSize) are not valid.

Description:

This function responds to the operation request from the application through the function call DeviceIoControl. The operation will depend on the I/O control code and its associated parameters.

5.2.5. DM Internals

The Disk Manager Internals will perform the functions of a switch wherein the various requests will be directed to different disk targets based on the system configuration. Two functions are required. One is the

generic queue function to manage the requests. Another is the dispenser function that directs the requests to various disk targets. To make the DM robust, it is desirable for the DM not to wait for the completion of a request. Rather, the DM could poll for the completion of the request, or the completed request is placed in a completion queue, or the DM Disk Interface completes the request via calls to functions exported by the DM Internals.

5.2.5.1. Queue Functions

The queue functions may derive directly from the EFI library with or without minor modifications.

5.2.5.1.1. dmCreateQueue

5.2.5.1.2. dmDeleteQueue

5.2.5.1.3. dmIsQEmpty

5.2.5.1.4. dmQInsertItem

5.2.5.1.5. dmQDeleteItem

5.2.5.1.6. dmQDeleteAll

5.2.5.2. Dispenser Functions

5.2.5.2.1. dmGetNextRequest

This function retrieves the next request from the request queue.

5.2.5.2.2. dmCompleteRequest

The function may be exposed to the disk service providers to complete a request.

5.2.5.2.3. dmDispatch

This routine will dispatch the requests based on disk I/O code and the assigned I/O disk service provider.

5.2.6. DM Disk Interface

The Disk Manager Disk interface provides the disk service provider interface.

5.2.6.1. dmEnumProvider

This function enumerates all the available disk service providers from the Disk Manager's disk service provider list.

5.2.6.2. dmFindProvider

This function searches for a specific disk service provider from the Disk Manager's provider list.

5.2.6.3. dmR gist rProvider

The disk service provider will submit the dispatching routine and potentially disk volume information for the disks it manages to the Disk Manager through this function.

5.2.6.4. dmUnregisterProvider

This function removes this disk service provider from the service provider list of the Disk Manager.

6. User Interfaces

No user interface is directly provided by the Disk Manager. Any configuration should be performed through the Supervisor Administrator mechanism.

6.1. Interface <name>

6.1.1. Overview

6.1.1.1. Purpose of Interface

6.1.1.2. Functions

6.1.1.3. Form

6.1.2. User Manual

6.1.3. Additional Information

6.1.3.1. Security and Integrity

6.1.3.2. Diagnostic and Debug Methods

6.1.3.3. Performance

6.1.3.4. Future Enhancements

7. System Software Interfaces

The system software interfaces include the interface between the Windows CE vDisk driver and the Disk Manager, and the interface between the Disk Manager and the disk service provider, which have been described in details in Section 5.

7.1. Interfac <nam >

7.1.1. Overview

7.1.1.1. Purpose of Interface

7.1.1.2. Functions

7.1.1.3. Form

7.1.2. Interface Description

7.1.3. Additional Information

7.1.3.1. Security and Integrity

7.1.3.2. Diagnostic and Debug Methods

7.1.3.3. Performance

7.1.3.4. Future Enhancements

8. Future Interfaces

9. Standards and Conformance

10. Release and Installation Procedures

11. Significantly Altered or Deleted Features

12. Design Summary

12.1. Design Overview

12.2. Work List of Required Implementation Activities

12.3. Design Hierarchy

13. Detailed Design

13.1. <module or s ction name> Detailed Design

13.1.1. Detail d Design of <module or proc dure name>

13.1.1.1. Interfaces

13.1.1.1.1. Entry Conditions

13.1.1.1.2. Exit Conditions

13.1.1.1.2.1. Normal Exits

13.1.1.1.2.2. Error Exits

13.1.1.2. Data References

13.1.1.2.1. Global Data

13.1.1.2.2. External Data

13.1.1.2.3. Internal Data

13.1.1.3. Source Language

13.1.1.4. Key Algorithms

13.1.1.5. Environmental Considerations

13.1.1.6. Reliability, Availability, and Serviceability

13.1.1.6.1. Error Avoidance

13.1.1.6.2. Error Detection

13.1.1.6.3. Error Recording and Reporting

13.1.1.6.4. Error Recovery

13.1.1.6.5. Error Analysis and Correction

13.1.1.7. Testing Considerations

13.1.1.8. Alternatives Rejected

14. Implementation Considerations

14.1.Usag Characteristics

14.2.Coding Constraints

14.3.Conventions and Standards

15. Performance

16. Integration

17. Migration

18. Issues and Risks

App ndix A. D finitions, Acronyms, and Abbreviations

This is a sample entry

Use the "Gloss" paragraph style. Place the term or expression to be defined on the first line. Then enter 1 or 2 line breaks (<SHIFT> <ENTER>)- not paragraph breaks. Next enter the definition. The key is to keep one entry as one paragraph. If this is done, you can select all of the glossary entries and use the Sorting function (under the Table menu) to alphabetize them.

App ndix B. Alternatives Consid red

App ndix C. Functional Design Summary

Requirement ID	Requirement	Commitment	Source	Function

App ndix D. D tail d D sign Summary

Requirement ID	Requirement	Commitment	Source	Design

Document made available under the Patent Cooperation Treaty (PCT)

International application number: PCT/US04/033253

International filing date: 07 October 2004 (07.10.2004)

Document type: Certified copy of priority document

Document details: Country/Office: US
Number: 60/509,581
Filing date: 08 October 2003 (08.10.2003)

Date of receipt at the International Bureau: 01 December 2004 (01.12.2004)

Remark: Priority document submitted or transmitted to the International Bureau in compliance with Rule 17.1(a) or (b)

World Intellectual Property Organization (WIPO) - Geneva, Switzerland
Organisation Mondiale de la Propriété Intellectuelle (OMPI) - Genève, Suisse